

Universitat Politècnica de Catalunya
Computer Architecture Department

Barcelona Supercomputing Center
Computer Sciences Department
Parallel Programming Model Group

User-directed Vectorization in OmpSs

Diego Luis Caballero de Gea

Master's Thesis

September, 2011

Directed by:

Xavier Martorell Bofill
Computer Architecture Department
Universitat Politècnica de Catalunya

Alejandro Duran González
Parallel Programming Models Group
Barcelona Supercomputing Center

Master's thesis presented in fulfilment of the
Master in Computer Architecture and Network Systems requirements

Abstract

In the recent shift to the multi-core and many-core era, where systems tend to be heterogeneous even at chip level, SIMD instruction sets and accelerators that exploit parallelism in a similar way are coming into prominence in new multiprocessors and systems. This heterogeneity, even at chip level, is causing a lot of trouble to compilers and parallel programming models in terms of being able to maximize the profitability of the computational resources in an easy, generic, efficient and portable fashion.

Although a lot of work on automatic vectorization/simdization techniques has been done over the years, compilers show important limitations when vectorizing code with pointers and function calls because of the traditional compiler analysis limitations, such as those in pointers aliasing analysis.

Concerning parallel programming models, some of them are restricted to specific architectures while other portable ones, such as OpenCL, require programmers to face low-level architecture details and hard source code transformations, presenting important performance problems among different architectures, which requires new tuning efforts.

In an attempt to offer a unified and generic solution to the auto-vectorization/simdization and portability problems, we propose *User-directed Vectorization in OmpSs*, a high-level programming model extension that offers developers the possibility to easily guide the compiler in the vectorization process just introducing some simple notations on the vectorizable areas of the code, such loops and functions.

We focused our particular design, implementation and evaluation on the Intel SSE instruction set for CPUs, getting the same or higher speed-ups than using the GCC compiler auto-vectorization in easily-vectorizable codes, and a performance improvement of up to 2.30 in more complex codes where GCC is not able to apply auto-vectorization and the hand-coded OpenCL version reaches a speed-up of 2.23.

Contents

Chapter 1: Introduction and Context	1
1.1 Introduction.....	1
1.2 Context.....	2
Chapter 2: State of the Art.....	5
2.1 Kinds of Parallelism.....	5
2.1.1 Instruction Level Parallelism (ILP)	5
2.1.2 Thread-Level Parallelism and Task-Level Parallelism (TLP).....	5
2.1.3 Loop-Level Parallelism (LLP) and Vector Parallelism.....	6
2.1.4 SIMD Parallelism	6
2.1.5 Other Kinds of Parallelism	7
2.2 SIMD Instruction Sets and Architectures	7
2.2.1 SIMD Streaming Extensions (SSE) Family	7
2.2.2 AVX and AVX2	9
2.2.3 LRBni	9
2.2.4 Altivec, VMX and Velocity Engine	10
2.2.5 Neon	11
2.2.6 Cray Traditional Vector Supercomputers.....	11
2.3 SIMD Parallel Programming Models and Extensions	12
2.3.1 CUDA.....	12
2.3.2 OpenCL	13
2.3.3 Array Building Blocks.....	13
2.3.4 User-mandated Vectorization in the Intel C++ Compiler 12.0	14
2.4 Related Work on Automatic Vectorization	14
Chapter 3: Motivation and Goals.....	17
3.1 Motivation.....	17
3.2 Goals	20

Chapter 4: Methodology	21
Chapter 5: Environment	23
5.1 OmpSs.....	23
5.2 Mercurium Compiler	24
5.3 Nanos++ Runtime	25
Chapter 6: Proposal: User-directed Vectorization.....	27
6.1 SIMD Directive.....	27
6.2 Implementation Design.....	29
Chapter 7: Design & Implementation.....	31
7.1 Design	31
7.1.1 SIMD Scheme on the Mercurium Compiler.....	31
7.1.2 Generic Vectors	33
7.1.3 Generic Functions.....	33
7.2 Generic SIMDization Phase Implementation	35
7.2.1 SIMD Directive	35
7.2.2 Generic Vector Data Type	35
7.2.3 Generic Vectorization Process	36
7.2.4 Generic Functions Infrastructure	37
7.3 SSE Code Generation Phase Implementation.....	37
7.3.1 SSE Vectorization Process	37
7.3.2 Vector Loads/Stores	38
7.3.3 Scalar Expansion	38
7.3.4 Conditional Operator	39
7.3.5 Operations on Mixed Data Lengths and Vector Conversions	39
7.3.6 Gather Operations.....	42
7.3.7 Architecture and Compiler Default Functions. ACML Support.	43
7.4 Limitations of the current implementation	43
7.5 CUDA and OpenCL approaches.....	44
Chapter 8: Experimental Results	45
8.1 Environment.....	45

8.2	Definition and Methodology of the Experiments	46
8.3	Benchmarks	47
8.4	Results.....	48
8.4.1	Saxpy/Daxpy	48
8.4.2	Vector-vector Multiplication	50
8.4.3	Matrix-matrix Multiplication.....	51
8.4.4	H264	52
8.4.5	Fast Walsh Transform	53
8.4.6	Blackscholes	54
8.4.7	Perlin Noise	56
8.5	Discussion.....	57
Chapter 9: Conclusions and Future Work		59
9.1	Conclusion	59
9.2	Future Work.....	59
References		61
Appendix A: Benchmarks Source Code.....		65
A.1	Blackscholes	65
A.2	Perlin Noise.....	68

List of Figures

Figure 1. Perlin Noise speed-up using GCC, ICC and OpenCL. Serial version	17
Figure 2. Generic compiler implementation scheme of SIMD directive	29
Figure 3. SIMD directive implementation scheme in Mercurium Compiler.	31
Figure 4. Generic representation of SIMD functions	34
Figure 5. Saxpy speed-up	48
Figure 6. Daxpy speed-up.....	49
Figure 7. Vector-vector Multiply single-precision speed-up.....	50
Figure 8. Vector-vector Multiply OpenCL single-precision speed-up.....	51
Figure 9. Matrix-matrix Multiplication single-precision speed-up	51
Figure 10. Matrix-matrix Multiplication double-precision speed-up.....	52
Figure 11. H264 single-precision speed-up	52
Figure 12. Fast Walsh Transform single-precision speed-up	54
Figure 13. Fast Walsh Transform double-precision speed-up.....	54
Figure 14. Blackscholes single-precision speed-up.....	55
Figure 15. Blackscholes OpenCL single-precision speed-up.....	55
Figure 16. Perlin Noise single-precision speed-up	56
Figure 17. Perlin Noise OpenCL single-precision speed-up	57

List of Tables

Table 1. Description of Generic Vectors Built-ins	36
Table 2. System hardware summary	45
Table 3. System software summary	45
Table 4. Compilation flags of each particular configuration and compiler	46

List of Listings

Listing 1. Perlin Noise Algorithm: Scalar (left) vs. OpenCL (right) versions	18
Listing 2. Perlin Noise ‘main’ function description: Scalar (left) vs. OpenCL (right)...	19
Listing 3. OmpSs example. Multi-architecture matrix-matrix multiplication.....	23
Listing 4. SIMD directive specification	27
Listing 5. SIMD directive usage example	28
Listing 6. SSE code (a) vs. GCC vector extensions (b).....	32
Listing 7. Representation of three generic vector variables in C/C++	33
Listing 8. Scalar array access (a) and its corresponding vector access (b).....	38
Listing 9. Constant (a) and an induction variable (b) SSE scalar expansion.....	38
Listing 10. SSE implementation (b) of a scalar conditional operation (a)	39
Listing 11. Two simple vectorizable loops that operates on mixed data lengths	40
Listing 12. Pseudo-code from vectorizing loops in Listing 11. (128-bit vectors).....	41
Listing 13. SSE Single-precision floating- point to unsigned-char vector conversion.	42
Listing 14. Gather operation using SSE	42
Listing 15. Fast Walsh Transform kernel	53
Listing 16. Blackscholes kernel: annotated scalar code	66
Listing 17. Blackscholes kernel: intermediate generic code	66
Listing 18. Blackscholes kernel: GCC vector extensions with ACML.....	68
Listing 19. Perlin Noise kernel: annotated scalar code	70
Listing 20. Perlin Noise kernel: intermediate generic code.....	71
Listing 21. Perlin Noise kernel: GCC vector extensions.....	76

Acknowledgements

I would like to take the opportunity to thank all the colleagues that helped and supported me in some way along this project.

Firstly, thanks to Barcelona Supercomputing Center, especially the Parallel Programming Models group from the Computer Sciences department, for choosing me for this exiting project and supporting me even in difficult times. Among them, the following ones deserve special mention: Xavier Martorell and Alejandro Duran, for their thorough review and valuable advice, always letting me propose and express my own ideas but keeping my feet on the ground; Roger Ferrer, for his infinite patience showing me the intricacies of the Mercurium compiler, standing my also infinite number of questions and compiler issues and always responding with professional and friendly attitude; Sara Royuela for her friendly support solving my doubts about techniques on compiler analysis and standing my insistent demands on new features in this fields.

In addition, I want to thank Dorit Nuzman (IBM Haifa) and her team, for sharing with us some of the benchmarks that they used in their publications related with auto-vectorization, and Lorena Horrillo, for giving to this Master Thesis a more diplomatic style.

We also thankfully acknowledge the support of the European Commission through the ENCORE project (FP7-248647), the Spanish Ministry of Education (TIN2007-60625 and CSD2007-00050) and the Generalitat de Catalunya (2009-SGR-980 and 2009-SGR-1372).

1

Introduction and Context

1.1 Introduction

Computer architects are nowadays facing new challenges in the way to improve performance and scalability in computation raised by the increasingly stringent constraints on frequency, area and power consumption. Current trends in processor architecture point towards systems with a high number of processing units, some of them heterogeneous with a specific purpose even at chip level.

As part of that heterogeneity, SIMD- and vector-alike units are playing an important role in new processor generations. These units aim to increase the data-level parallelism that can be exploited in applications. For example, SSE, VMX and Neon SIMD extensions, from Intel, IBM and ARM respectively [2, 4, 7], support vectors that are able to work on up to four single floating-point elements at a time (128 bits). AVX and XOP extensions [9, 10], recently incorporated in the latest Intel and AMD processors respectively, extend this number to eight single floating-point elements (256 bits), and future extensions of 512 (LRBni) are under prototyping [12, 17]. This trend shows the increasing interest in exploiting parallelism this way.

Furthermore, with due respect to the inevitable differences, some accelerators such as GPUs have a similar behaviour when they exploit parallelism creating thousands of threads. These threads are clustered in groups and all threads in a group tend to execute the same instruction at the same time on different data. As a result, each set of threads can be seen as a *special* vector unit exploiting parallelism in a SIMD way on a data vector.

As a consequence of this new era of heterogeneous multi- and many-core systems, exploiting the full computational power of each single component is now more important than ever. Sometimes programmers tend to focus only on the benefits of the multi-threading programming, forgetting the impressive potential that a single core can offer and the substantial improvement that a better profitability of them can lead in a parallel application.

This idea is particularly important in the last computational trends, like Cloud Computing, where the main goal is to maximize the profitability of energy and

computational resources. In fact, some developers have already begun to realize that SIMD capabilities are underutilized in this environment [13].

Nevertheless, compiler technology is still far from being able to efficiently exploit SIMD hardware in a generic way. For example, codes that use pointers usually cause the compiler not to know if the accessed data will be properly aligned or if it will be aliased to other references. Also in the presence of function calls, particularly when the source code is not accessible, usually the compiler inhibits itself from vectorizing because it does not know if an equivalent vector function exists.

In addition, in heterogeneous architectures where each component may have its particular characteristics or different instruction sets to achieve the same goal, the compiler would need to be aware of all the specific low-level details, manage them and decide what the most suitable processing unit is for the execution of a particular region of code. Otherwise, the resulting code will get poor performance.

In an attempt to help the compiler and ease SIMD profitability, some programming models and libraries with SIMD support, such as CUDA, OpenCL, Intel ABB [14, 15, 16], emerged. However, while some of them are specific for particular architectures, others expose too many low-level details and require hard source code transformation, which make them not useful or efficient for the vast majority of programmers.

To address these issues, we present a proposal that allows programmers to efficiently direct the compiler in the vectorization process by means of a generic and simple annotation on those loops and functions that can be vectorized.

The rest of the document is organized as follows: Chapter 2 presents the state of the art in SIMD instruction sets and programming models. Chapter 3 describes the motivation and goals of the project. Chapter 4 defines the methodology followed along the whole project. Chapter 5 describes the environment of the project. Chapter 6 introduces the User-directed Vectorization proposal. Chapter 7 contains the design and implementation details. Chapter 8 shows the evaluation results and Chapter 9 concludes and delineates the future work. Finally, one appendix contains a summary of the benchmarks source codes, presented in the different states of our transformations.

1.2 Context

The current project is conducted as Master's thesis in Computer Architecture, Networks and Systems (CANS), at the Technical University of Catalonia (UPC), and has been funded by Barcelona Supercomputing Center (BSC) and the European Commission through the EnCORE project (FP7-248647).

The project has been developed as resident student in the Programming Models group of the Computer Sciences department at Barcelona Supercomputing Center. The main goal of this group is to research on current and new programming model paradigms focused on high-performance computing and general-purpose parallelism from the point of view of expressiveness, usability, efficiency and profitability of the computational power of the different architectures, ranging from SMP processors and accelerators, to shared- and distributed-memory systems.

The programming model group is proposing extensions to OpenMP to improve the expressiveness of the model and prove it with support for the latest accelerator architectures. They have their own parallel programming model for heterogeneous architectures, OmpSs, and implement their solutions and prototypes on the Mercurium C/C++ source-to-source compilation infrastructure and the Nanos++ runtime infrastructure.

This project is framed in the context of two local projects: the OmpSs project and the Mercurium project. We are extending the OmpSs parallel programming model with support for SIMD architectures, whereas the corresponding implementation of the prototype is developed in the Mercurium compiler.

State of the Art

2.1 Kinds of Parallelism

Many different definitions of parallelism have emerged over last decades. Some of them have a transversal meaning, although others keep a certain similarity among each other or they form a subset of a more generic one.

In this section we briefly describe some of the most important kinds of parallelism that are in some way closely related to this project.

2.1.1 Instruction Level Parallelism (ILP)

Instruction Level Parallelism (ILP) [18] is the parallelism automatically exploited by superscalar processors, which have a pipelined execution flow to improve performance overlapping the execution of instructions. This overlap may potentially result in the parallel execution of different instructions without, supposedly, the intervention of programmers. In practice, loops are the most important source of ILP since unrolling them several iterations could offer plenty of independent instructions and sometimes, programmers have to apply some transformation on the source code to enhance the further exploitation of this kind of parallelism.

ILP is also exploited in VLIW processors, where the compiler plays an indispensable role in order to reorder and pack instructions that can be executed in parallel.

This sort of parallelism was an active topic in research [19, 20] until the salient shift to the multi- and many-core era in the past few years, when other types of parallelism, like Thread-Level Parallelism and Data-Level Parallelism, came into prominence.

2.1.2 Thread-Level Parallelism and Task-Level Parallelism (TLP)

These two kinds of parallelism are very close to each other. The term Thread-Level Parallelism [18] arose with the advent of MIMD (Multiple Instructions streams, Multiple Data streams) multiprocessors, which are systems where each single processor may execute its own instruction stream on a different data set at the same time. If this

multi-stream parallel execution is extrapolated within the scope of a single application, each flow of execution is called *thread* and the parallelism is named as Thread-Level Parallelism. There are technologies like Simultaneous Multithreading [21] that allow the concurrent execution of multiple threads on a single processor or core.

On the other hand, Task-Level Parallelism, also known as *function parallelism*, is another sort of parallelism that could be seen as a high-level subset of Thread-Level Parallelism which emphasizes more on differentiating and delimiting the different parts of code that will be synchronously or asynchronously executed as an isolated entity of the program. Task-Level Parallelism is exploited in very important parallel programming models, like OpenMP, Intel TBB, CUDA and OpenCL [22, 23, 14, 15], and it is becoming an outstanding topic in nowadays research [24, 25, 26].

2.1.3 Loop-Level Parallelism (LLP) and Vector Parallelism

This level of parallelism, also known as Data-Level Parallelism (DLP), is present in loops where iterations can be executed in parallel [27]. This way, data structures involved in the computation can be distributed across different computational units and be computed in a parallel fashion. In its most generic description, there are no restrictions on data types as long as iterations can be executed anyway in parallel, so this kind of parallelism is generically exploited by MIMD machines [28].

A subset of this level of parallelism is the Vector Parallelism that can be exploited by traditional vector machines. In this case, loop data structures need to satisfy the particular vector machine constraints, like certain access patterns, so there exists loops with LLP that cannot be executed in this sort of architectures.

2.1.4 SIMD Parallelism

SIMD (*Single Instruction Multiple Data*) Parallelism [18, 28] emerged with the rise of massively parallel supercomputers, like the Illiac IV [29]. Normally, these systems consisted of a control processor and a huge array with thousands of data processing elements where each one was traditionally interconnected directly to its neighbours. This way, the control processor broadcasted the same instruction to the processing elements that executed it on different data items. Due to the interconnection topology, this kind of systems suffered important problems when data movements took place among processing elements not closed each other, which finally ended causing important restrictions on applications that could be executed efficiently, even tighter than those from vector or MIMD architectures.

However, nowadays this designation has been commonly adopted to refer to *Single-Instruction-Multiple-Data* parallelism in a more generic and less restricted way. Some current multiprocessors use *SIMD* and *vector* terms to describe their sets of

instructions that can operate on different data elements at a time, although SIMD is usually used to denote a more generic and unrestricted way of parallelism.

2.1.5 Other Kinds of Parallelism

There are some new definitions of parallelism that have emerged in the last years and have had some acceptance and impact in the research community. The most significant two are *Superword Level Parallelism* (SLP) and *Single Instruction Multiple Threads Parallelism* (SIMT).

SLP [28] is defined to differentiate traditional large-scale SIMD parallelism from the small-scale parallelism that can be exploited by the latest SIMD extensions from current multiprocessors. Since this new kind of parallelism is focused on SIMD operations on a small number of data elements, this approach also allows to efficiently exploit parallelism packing similar operations when the number of them is not very high.

SIMT is the term apparently created by NVIDIA [14] that aims to establish certain distance between the already known SIMD/Vector Parallelism and the parallelism exploited by their GPUs and their CUDA programming model. The key difference is “that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behaviour of a single thread.” This means that programmes must have in mind that they are writing code for independent parallel threads rather than for vector registers.

2.2 SIMD Instruction Sets and Architectures

In this section, we briefly describe the most salient SIMD instruction sets that are part of nowadays general-purpose multiprocessors and embedded systems.

2.2.1 SIMD Streaming Extensions (SSE) Family

Streaming SIMD Extensions (SSE) [2, 3] is the base name of the largest Intel SIMD instruction set family that extends the traditional x86 architecture with SIMD instructions that may operate on several scalar elements at a time. Up to six different subsets have been released over the recent years: SSE (1999), SSE2 (2001), SSE3 (2004), SSSE3 (2005), SSE4.1 and SSE4.2 (2006/2007).

The SSE family is the 128-bit successor of the 64-bit **MMX** instruction set. The first SSE release is only able to work on integer data types and it shows some problems because of the floating-point registers re-use for vector purposes.

The first generation of **SSE** added 70 new instructions that work on a new eight-register set (XMM registers) divided into four 32-bit single-precision registers (shared). They are mainly focused on single-precision floating-point data types. Thus, the same

operation can be performed on four 32-bit floating-point elements at a time, with the consequent performance improvement.

In addition to data type restrictions, two important constraints need to be taken into account when using SSE:

- Target data elements must be contiguous in memory (packed) in groups of the vector size (16 bytes). If they are not, programmers will have to reorganize them in this way because SSE instructions cannot work on disjoint data elements.
- The first packed element has to be properly aligned to 16 bytes. If not, special and slow load/store instructions must be explicitly used or the execution will fail.

For these reasons, the usefulness of the first version of SSE was limited to very specific areas, like digital signal processing and multimedia.

SSE2 introduced 144 new instructions that mostly added support for integer and double-precision floating-point data types, increasing the scope of applications that may benefit from them. New integer instructions aimed at fully substituting MMX technology extending the instruction set to 128-bit XMM registers. Later implementations of SSE2 in x86_64 architectures also increased the register set from 8 to 16 registers.

Three years later, **SSE3** was released with 13 new instructions targeting to improve horizontal operations among the scalar elements of the same vector register. Some of them essentially reduce pairs of elements applying an addition or subtraction between them, or even they perform several simple operations at a time. A few more instructions are designed to improve subprocesses synchronization and misaligned memory loads.

SSSE3 (Supplemental Streaming SIMD Extension 3) emerged a year after with 16 new instructions that are able to work on both MMX and XMM registers. They complement SSE3 instructions with support for new data types and add some others that implement absolute value and shuffle operations.

SSE4 is the latest update on SSE and supposes a great advance towards generic-purpose computation. The first set of instructions was called **SSE4.1** and consists of 47 new instructions with a wide functionality: from specific instructions that perform the dot product and maximum/minimum operations, to conditional copies, floating-point rounds and instructions to easily insert and extract scalar elements from vector registers. The second instalment, **SSE4.2**, incorporates other 7 instructions mainly focused on string manipulation, really useful in word processors, and instructions to compute the Cyclic Redundant Check (CRC32).

AMD processors have been keeping up-to-date in this area including support for most SSE instructions and developing their own vector extension concurrently to Intel extensions, like **SSE4a** [10].

2.2.2 AVX and AVX2

Advanced Vector eXtensions (AVX) [9] is the name of the brand new Intel SIMD instruction set extension that seeks to replace the previous Streaming SIMD Extensions generation. This new instruction set is mainly focused on improving the performance of floating-point operations in the same way that SSE (single-precision) and SSE2 (double-precision) did in the past, but in this case, AVX extends vector registers and floating-point instructions to 256 bits.

AVX adds 35 256-bit single/double-precision floating-point arithmetic instructions ranging from the typical addition and subtraction to max, min and round operations. It also increases the number of conditional operations to 32, improving programming flexibility. 39 non-arithmetic instructions are promoted to 256 bits from SSE, such as logical, blend, conversion, test, unpacking, shuffle, load and store operations. Other 18 new advanced 256-bit data processing instructions are incorporated supporting conditional load/stores, intra-register manipulation (new permutation and insert/extract primitives) and new blend operations.

In addition to the widening of the vector registers and new instructions, AVX introduces a new more flexible three-operand instruction syntax and encoding, looking for flexibility and thinking in further extensions with wider vector registers and instructions with up to four instructions operands.

In conjunction with AVX, an additional SIMD extension, FMA, adds a wide variety of *fused-multiply-add* operations.

AVX2, the second release of this SIMD instruction set, was briefly announced for Intel's new generation processors, in 2013. It promises to incorporate 256-bit integer instructions as well as support for new features such as broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

For its part, AMD follows a similar line in the sense of extending its 128-bit SIMD instruction sets to 256 bits and providing new functionality. XOP and FMA4 [10] extensions are proof of that. They implement a bit similar features than AVX and FMA although there are instructions with functionality not supported in both directions.

2.2.3 LRBni

LRBni [11] is a new Intel SIMD instruction set that appeared with the advent of the Larrabe architecture [12]. It is developed in parallel with SSE and AVX instruction sets

and it is characterized by a 512-bit wide integer and single/double floating-point vector instructions.

Larrabe vector instruction set reaches new milestones in flexibility and SIMD general-purpose offering a complete instruction set that is able to perform gather/scatter memory operations, a large number of transcendental math functions, horizontal operations like scalar reductions (using different arithmetic operations) and a plenty of masked instructions, among other things.

2.2.4 Altivec, VMX and Velocity Engine

Velocity Engine, Vector/SIMD Multimedia eXtensions (VMX) and Altivec are the three different names that nominate the same SIMD instruction set owned by Apple, IBM and Motorola (AIM alliance), respectively, released in 1999 [30, 4, 5].

This vector processing technology defines 128-bit vector instructions that operate on single-precision floating-point and integers data types. They were specially designed, as many others, to improve performance in multimedia application and are present in several scenarios, like high performance computing, multimedia stations and embedded systems.

From the beginning, **VMX** had its own vector register file composed of 32 vector registers architecturally separated from the floating-point and the general-purpose register files. The instruction set came with a very complete functionality since the first version, unique at that time. In addition to the common arithmetic instructions on different data types, it introduces support for vector comparisons and select mechanisms to address the execution of conditional code in a SIMD manner. It also provides an important set of horizontal operations to work on the scalar elements within a vector and adds an innovative flexible vector permutation instruction that is able to perform sophisticated data manipulations.

IBM vector instruction set has only been publicly updated two times, although it seems that the corporation continues actively researching on this topic [Referencia web IBM]. The first VMX update is **VMX128** [31]. It customizes VMX for use in the Xeon processor (Xbox 360) adding support for some operations very common in graphics applications, like dot product, and increasing the number of vector registers from 32 to 128, among other features.

Vector-Scalar eXtension (**VSX**) [6] is a more generic-purpose second update that extends VMX with support for new data types, like double-precision floating-point, and introduces a unified register file of sixty-four 128-bit registers (Vector-Scalar Registers) for scalar and vector instructions that allows a fast data migration between them without memory overhead.

2.2.5 Neon

Neon (Advanced SIMD) [7, 8] is the name of the vector technology in ARM architectures that performs SIMD operations on 64-bit and 128-bit vector registers.

This instruction set supports operations on integer (8, 16, 32 and 64 bits) and single-precision floating-point data types in a vector manner, meanwhile double-precision floating-point operations are executed in a scalar fashion. It also has a special data type to perform operations on polynomials and the half-precision floating-point type is also supported in some implementations.

Neon technology is highly specialized and focused on multimedia applications and signal processing algorithms such as video and audio decoding/encoding, 2D/3D graphics, gaming, image processing, telephony, etc. For this reason, this instruction set provides a rich number of instructions that performs the most common operations in this field. Some remarkable ones are some interesting blends operations, advanced comparison instructions such as those that perform the absolute value before the comparison, tests, shuffles (vector elements transposition), vector packing/unpacking (interleaving/de-interleaving), instructions that perform arithmetic operations and shift the resulting value, horizontal operations, load/store with de-interleaving/interleaving, etc.

We can find this technology implemented in all the current ARM Cortex-A CPUs and Mali graphic processing units, among others.

2.2.6 Cray Traditional Vector Supercomputers

Seymour Cray's architectures [18, 32] can be considered as the parents of traditional vector computation. Although it was not the first vector computer, the first Cray-1 system started a transcendent revolution in supercomputing that still has its effects in nowadays systems.

Cray-1 (1976) was a vector supercomputer with eight 4096-bit vector registers (64-bit elements), with 6 vector arithmetic units (FP add, FP multiply, FP reciprocal, integer add, logical and shift) and one special unit for memory loads/stores. All vector operations were performed between registers (vector-register processors) in contrast to memory-memory vector processors. Since this first version, Cray supercomputers supported conditional operations by means of bitwise masks, and constant non-stride-one memory data accesses.

Cray X-MP (1983), Cray Y-MP (1988) and Cray-2 were the next generation of supercomputers with the same Cray-1 vector philosophy, although the first two were multiprocessor systems. Cray Y-MP was the first supercomputer to reach up to 2.3 sustained gigaflops.

Other award-winning vector supercomputers were Cray C90 (1991) and Cray T90 (1995), with eight 8192-bit vector registers, establishing new peak performance and pioneering supercomputer wireless technology, respectively.

New supercomputer systems continued appearing later, but multiprocessors technology ended up imposing on them because of their cheapest design and production process and their more general-purpose usefulness.

Nowadays, Cray is still successfully working on the supercomputer area but their systems rely on Intel and AMD multiprocessors and even in NVIDIA GPUs, like Cray XK6, Cray XE6 and Cray CX 1000 systems or the famous Jaguar, 2009/2010 1st top500 supercomputer, based on Cray XT5.

Cray SV1 (1998), Cray SV1ex (2001) and Cray X1E (2003) were the last Cray supercomputers that still maintain traditional vector characteristics in some way.

2.3 SIMD Parallel Programming Models and Extensions

In this section we describe some parallel programming models and extensions that tend to exploit parallelism in a SIMD way.

2.3.1 CUDA

CUDA[14] stands for Compute Unified Device Architecture and it is the architecture and programming model challenge of NVIDIA Corporation for deploying its graphic processing units (GPUs) on the world of General-Purpose computing on Graphic Processing Units (GPGPU).

Although we cannot strictly classify CUDA as a SIMD programming model (See section 2.1.5), it exploits parallelism in a very similar manner. Workloads are described using kernels that generically contain the behaviour of a single GPU-thread. Then, these kernels are set up with the number of threads that will execute the kernel code on the GPU. This threads configuration is described using two levels of abstraction. The first level groups threads in blocks (fine-grained level) and the second one determines the total number of blocks involved in the kernel execution (coarse-grained level).

The main execution flow consists of one (or multiple) thread running in the CPU in order to manage and launch these kernels to the GPU.

The programming language on which this programming model is based is C++ with some particular extensions that enrich the description related with the GPU philosophy. CUDA offers two application user interfaces with important differences in the programming abstraction level. The first one, CUDA Driver API, is a low-level interface that lets advanced developers to control very specific-architecture details. On the other hand, the CUDA Runtime API, provides a higher level of abstraction.

2.3.2 OpenCL

OpenCL [15] is an open standard that pretends to define a general-purpose unified parallel programming model that solves the problem of programming for heterogeneous systems with accelerators and the associated execution portability among them. Thus, the main goal of OpenCL is to offer “portable and efficient access to the power of these heterogeneous processing platforms”, i.e., to allow programmers execute the same code on different architectures with the minimum number of source code changes, if any, ranging from personal computers to servers and embedded systems.

The OpenCL parallel programming model is based on C99 with specific extensions and it is able to exploit both Data-Level Parallelism and Task-Level Parallelism. It bears some resemblance to the CUDA parallel programming model in the sense that workloads are also structured in two level of abstraction: work-items (fine-grain) and work-groups (coarse-grain). They also have a similar low-level application programming interface (Driver API in CUDA) that provides access to low-level details of the architecture which also results in a more complex programmability.

Several OpenCL implementations are based on a Just-In-Time (JIT) compiler that carries out a lightweight architecture-specific compilation phase at runtime, providing the aforementioned portability execution. However, the most pronounced issue in OpenCL is the poor performance portability [33, 34, 35], which means that OpenCL applications that get good performance in some architectures do not get such a good performance in others, requiring new changes in the source code.

In practice, OpenCL is being more successful in multimedia and gaming sectors than in scientific and supercomputing ones.

2.3.3 Array Building Blocks

Intel Array Building Blocks (ArBB) [16] is a brand new data-parallel programming model aimed at offering a scalable and portable solution for the profitability of current and upcoming Intel SIMD multi- and many-core architectures. It is based on a C++ library interface that provides new impressive data types and operators, and a runtime which performs an online compilation phase that retargets the code to the underlying architecture.

Since ArBB is mainly a C++ library, developers can include it as part of their traditional C/C++ applications and compilation frameworks, only accelerating those portions of code that have more data-level parallelism potential.

For this purpose, ArBB defines new data types and operators that the programmers must use in order to accelerate such regions of code. We can find an equivalent definition for the traditional scalar data types, but the most notorious incorporation consists of a new kind of compound data types or data structures named *containers* which allow programmers to express operations at an aggregate data

collection level, such as images, matrixes or arrays, with its associated parallel operators. These containers can also be defined as dense containers or nested containers which will affect to the internal data representation and the parallel operations on them, all transparently to the user.

However, although these new features improve the expressiveness of the parallel programming model, programmers will have to recode those portions of code that they want to accelerate, facing challenging source code transformations or normally rethinking the whole algorithm in order to be able to incorporate the ArBB data structures and operators.

2.3.4 User-mandated Vectorization in the Intel C++ Compiler 12.0

The latest version of Intel C++ Compiler, released in November 7, 2010, implements an approach really close to our proposal [36]. They introduce several directives that allow programmers to guide the compiler in the auto-vectorization process annotating those innermost loops statements and functions candidates to be vectorized.

In order to tell the compiler that a for-statement is vectorizable, we should annotate such loop with the SIMD directive notation, `#pragma simd`, meanwhile `__declspec(vector)` is the annotation necessary to express the same meaning with respect to vectorizable functions.

Both directives also include several clauses that provide more expressiveness in the way of specifying architecture details, such the vector length, or harder operations, like reductions.

The most important differences from our approach is that Intel's is designed for their own architectures meanwhile we pretend to build a generic proposal that can be applied on any SIMD architecture and compiler, beyond a simple design and implementation for a particular one. In addition, we pretend to extend our proposal to more generic SIMD codes that for now are not vectorizable with this version of the Intel Compiler.

This compiler extension will be very valuable for the evaluation of our proposal.

2.4 Related Work on Automatic Vectorization

Automatic vectorization has been a well-studied topic in compilers research along the years since the traditional vector processor era. Current research in this field focuses its efforts on taking advantage of the short vector/SIMD extensions present in the latest multiprocessors.

Thus, we can find even the definition of *Superword Level Parallelism*, a new level of parallelism specific to these short SIMD instruction sets [28] which was the

basis of new related publications.

One of the most important issues that limits the auto-vectorization and performance of these kinds of architectures is the alignment constraints that the memory references of the source code need to satisfy. This problem has been studied several times throughout the last years, reaching solutions mainly based on the realignment of data at runtime or using special instructions for unaligned load/stores [41, 42, 47].

Beyond the simple auto-vectorization process of regular source code, in an attempt to promote the vectorization of more generic applications, some researchers faced more challenging situations, trying to vectorize codes where the execution flow can diverge even within the scalar elements of the same vector [51, 43]. This vectorization is carried out by means of predicated and select instructions.

Another exiting issue is the vectorization of codes with non-stride-one data accesses when the architecture has special support for them [48] or when there are no gather and scatter memory instructions available [50], where interesting data reorganization schemes are proposed to properly pack and unpack the disjoint data elements.

Some publications also address the vectorization process from the point of view of the outermost loop, taking advantage of the data-level parallelism at this level applying the unroll-and-jam technique [49].

We also can find publications to deal with SIMD instruction sets in specific contexts, such as the vectorization of the Fast Fourier Transform, where scalar elements permutations within vectors have an essential role [45].

The most recent publications implements much more complete and powerful auto-vectorization frameworks that address several issues of previous publications [43], or even multi-platform solutions dealing with alignment constraints and facing more advanced vector operations, such us reductions and partial sums [47]. Some of them make use of just-in-time compilers to bring portability to the vectorization process, introducing a lightweight compilation phase that adapts the code to the underlying architecture at runtime [46].

Our proposal is partially based on the concept of *virtual vectors*, introduced by an approach with a first vectorization phase aimed at vectorizing codes in a generic way without introducing specific architecture constraints, dealing also with mixed-length data elements [40, 42].

Motivation and Goals

3.1 Motivation

As introduced previously, SIMD extensions are becoming more important and play an essential role in aspects like performance and energy consumption. Furthermore, new trends in computation point towards extending SIMD capabilities to more general-purpose applications with more flexible instructions (See sections 2.2.2 and 2.2.3).

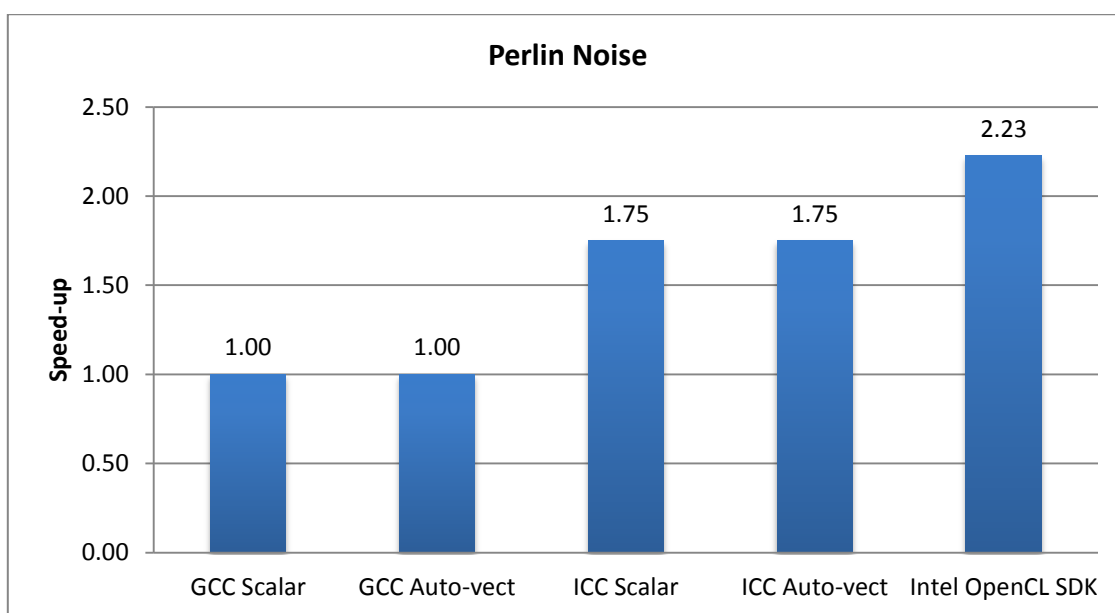


Figure 1. Perlin Noise speed-up using GCC, ICC and OpenCL. Serial version

However, from the compilers point of view, automatically taking advantage of these extensions is still a challenge. Figure 1 shows the achieved speed-up using the GNU compiler (GCC) and the Intel Compiler (ICC) auto-vectorization techniques on the Perlin Noise algorithm against a hand-coded OpenCL implementation, all of them using only one CPU thread. We can see how GCC completely inhibits itself from auto-vectorizing, offering the same speed-up than the scalar version. For its part, the ICC scalar execution reaches a speed-up of 1.75 over GCC as result of applying better optimizations, but auto-vectorization provides no performance improvements. In the

case of OpenCL, it gets a remarkable 2.23 speed-up factor, which is a really good result taking into account that we are using a single core.

A similar behaviour may be extrapolated to the vast majority of general-purpose applications, where compilers have to deal with complex source codes with pointers and functions calls. Thus, in an attempt to exploit the whole computational power of a particular architecture, programmers are forced to go through the arduous and error-prone process of recoding applications using OpenCL or any other low-level specific extension/language.

<pre>1 float noise3(float x, float y, 2 float z) 3 { 4 float floor_x = floor(x); 5 ... 6 int X = (int) floor_x & 255; 7 ... 8 int A = perm[X] + Y; 9 int AA = perm[A] + Z; 10 ... 11 } 12 13 void compute_perlin_noise(14 unsigned char * output_red, 15 unsigned char * output_green, 16 ...) 17 { 18 for (j = 0; j < img_height; j++){ 19 for (i = 0; i < img_width; i++){ 20 ... 21 red = noise3(xx, vt, yy); 22 ... 23 red = (red>1.0f) ? 1.0f : red; 24 ... 25 output_red[(j * rowstride) + i] 26 = (unsigned char)red; 27 } 28 } 29 } 30 31 32 33 34 35 36 37 38 39 40</pre>	<pre>1 /* Vector load of disjoint elements 2 int4 p(int4 i){ 3 int p0 = perm[i.x]; 4 int p1 = perm[i.y]; 5 int p2 = perm[i.z]; 6 int p3 = perm[i.w]; 7 int4 ret = (int4) (p0, p1, p2, p3); 8 return ret; 9 } 10 11 float4 noise3(float4 x, float4 y, 12 float4 z) 13 { 14 float4 floor_x = floor(x); 15 ... 16 int4 X = convert_int4(floor_x)&255; 17 ... 18 int4 A = p(X) + Y; 19 int4 AA = p(A) + Z; 20 ... 21 float4 g0 = grad(p(AA), x, y, z); 22 ... 23 } 24 25 __kernel void compute_perlin_noise(26 __global uchar4 * output_red, 27 __global uchar4 * output_green, 28 ...) 29 { 30 i = get_global_id(0); 31 j = get_global_id(1); 32 ... 33 red = noise3(xx, vt, yy); 34 ... 35 red += 0.35f; 36 ... 37 output_red[(j * rowstride) + i] 38 = convert_uchar4_sat(red); 39 ... 40 }</pre>
---	---

Listing 1. Perlin Noise Algorithm: Scalar (left) vs. OpenCL (right) versions

Listing 1 and Listing 2 briefly illustrate some of the most important changes applied on the Perlin Noise algorithm in order to be ported to OpenCL starting from the scalar version.

In Listing 1, we can see how a new function is necessary to perform vector load operations on disjoint elements. In addition, explicit function calls, like *conver_int4* and *convert_uchar4_sat*, are required to perform conversions when scalar types are replaced by OpenCL vector data types, and some operations, like conditional ones, are supplanted by other equivalent looking for getting a better performance under this programming model.

<pre> 1 int main (){ 2 // Memory allocation (CPU) 3 // Data Initialization (CPU) 4 5 // Perlin Noise Execution 6 compute_perlin_noise(output_red, 7 output_green, 8 ...); 9 10 // Memory Deallocation (CPU) 11 } 12 13 14 15 16 17 18 19 20 21 22 </pre>	<pre> 1 int main (){ 2 // Create OpenCL Device & Context 3 // Create Command Queue 4 // Create & Compile the Program 5 // Host Memory Allocation 6 // Host Data Initialization 7 // Device Memory Allocation 8 // Copy Data from Host to Device 9 // Setup Kernel Parameters Values 10 11 // Configure Kernel Dimensions 12 global_size[0] = img_width / VEC_WIDTH; 13 global_size[1] = img_height; 14 local_size[0] = local_work_group_size; 15 local_size[1] = 1; 16 17 // Perlin Noise Execution 18 clEnqueueNDRangeKernel(...); 19 20 // Copy Data Back from Device to Host 21 // Release Resources 22 } </pre>
---	---

Listing 2. Perlin Noise ‘main’ function description: Scalar (left) vs. OpenCL (right)

Listing 2 compares the *main* function of the scalar and the OpenCL algorithms and only names the different phases that take part in the last one, for simplicity. As shown, there are noticeable differences in complexity between the two versions, even though the specific code is not displayed in OpenCL. Several low-level details have to be inevitably managed by programmers with the associated effort.

Unfortunately, these efforts could be even higher if portability among architectures is required, because they will have to be recoded over and over again for each target architecture. OpenCL was released in order to mitigate the portability problem, but despite the fact that applications may run on different architectures, even without offline recompiling, performance portability is a big issue, being imperative significant changes in the source code if performance among architectures is a goal.

For these reasons, there is no doubt that it is necessary to research on a new approach that instructs compilers to automatically vectorize codes for different architectures and accelerators, focused on promoting performance portability, bypassing this way the well-known problems in compilation analysis that impede compilers to know whether a particular code is truly vectorizable.

Some programming model standards, like OpenMP, are actively working to improve productivity using these extensions and accelerators, so we believe we have an excellent opportunity to conduct research on this area that may lead to an interesting proposal that might be included in the next generation of programming models.

3.2 Goals

The main goals of this project can be summarized in the following five points:

- Take advantage in an effective way of the current and future SIMD processor extensions.
- Propose programming model support to drive the compiler using such extensions.
- Demonstrate ease of use of the proposed programming extensions.
- Evaluate the proposal with benchmarks known to be easy and difficult to *simdize* (exploit SIMD extensions).
- Compare the proposal with the related work.

Our intention is to start developing a proposal that might be included as part of the next version of the OpenMP standard, where accelerators have the leading role.

From the design and implementation point of view, this Master's thesis, with its corresponding limitation in time, aimed at creating a new compiler infrastructure that serves as basis for further development and prototypes in this field, always from the standpoint of research and not from production.

4

Methodology

This chapter describes the methodology that we have followed during the whole development of the project.

We began evaluating several ideas for this Master's Thesis. Once the topic was selected, the first phase of the project consisted of a deep evaluation of the related work and bibliography. We widely studied several SIMD architectures and instructions sets, auto-vectorization techniques, parallel programming model and compilers with SIMD support to realize where exactly were the challenges and limitations that we had to face.

We followed a spiral approach in the definition of the generic proposal and the particular design, implementation and evaluation that we have done in this project.

We started to build the first sketch of the proposal at the time that we were studying the Mercurium compiler features to know well the technology that it could offer at that moment and its limitations.

We continued with a first coarse-grained design step and the initial plan of the implementation of certain functionality in Mercurium that would be necessary for the project. Once the basis were designed and implemented, we moved to the design and implementation of the specific SIMD features.

From the design, implementation and evaluation of each feature we got the corresponding feedback that helped us in the more precise definition of the proposal and the following design and implementation of the next feature.

The project time constraints forced us to delimit the SIMD functionality that would be included in this particular design and implementation. In order to have a consistent set of features that could be successfully evaluable, we decided to focus on the vectorization of some benchmarks that we would use later for such evaluation, addressing the different issues always from a generic point of view and not from the particular case of the targeted benchmark.

5

Environment

This dissertation is developed in the context of the related projects OmpSs, Mercurium and Nanos++ and they three comprise the environment of this project.

5.1 OmpSs

OmpSs [57, 53] is a highly expressive parallel programming model, developed by the Barcelona Supercomputing Center (BSC), which aims to cover both homogeneous and heterogeneous architectures unifying them under the same programming model to minimize the programmability effort.

```
1  const int NB = 512;
2  #pragma omp task inout([NB*NB] C) input([NB*NB] A, [NB*NB] B)
3  void matmul_block(float * A, float * B, float * C){
4      // plain C kernel code for the SMP environment
5  }
6  #pragma omp target device(cell) copy_deps implements(matmul_block)
7  void matmul_block_cl(float * A, float * B, float * C){
8      // OpenCL kernel code
9  }
10 #pragma omp target device(cuda) copy_deps implements(matmul_block)
11 void matmul_block_gpu (float * A, float * B, float * C){
12     // CUDA kernel code
13 }
14 void matmul (int mDIM, int lDIM, int nDIM, float ** A, float ** B, float ** C){
15     for(i = 0; i < mDIM; i++) {
16         for (j = 0; j < nDIM; j++) {
17             for (k = 0; k < lDIM; k++) {
18                 matmul_block (A[i*lDIM+k], B[k*nDIM+j], C[i*nDIM+j]);
19             }
20         }
21     }
22     #pragma omp taskwait
23 }
```

Listing 3. OmpSs example. Multi-architecture matrix-matrix multiplication

It is based on OpenMP [22] and the task-oriented StarSs programming model extensions [26], which provides runtime dependence analysis among tasks, automatic task execution schedule, automatic data transfers between CPUs and accelerators and support for raw OpenCL and CUDA kernels as a solution to simplify and make programmability easier.

OmpSs is currently operating on Shared-Memory Processors (SMP), SMP with GPUs and clusters of both SMPs and GPUs (support for Cell B.E. is deprecated). Listing 3 depicts a brief example of a task-oriented matrix-matrix multiplication implementation with support for multiple architectures, such as Cell B.E., CUDA and SMP. Depending on the underlying architecture, the runtime will determine which version of the algorithm is the most appropriated to be executed.

5.2 Mercurium Compiler

The Mercurium compiler [55] is a source-to-source compiler with support for C, C++ and Fortran programming languages. It is developed by the Barcelona Supercomputing Center with the intention of building an infrastructure for the fast prototyping of parallel programming models, offering also automatic support for their Nanos++ runtime.

Mercurium has a wide experience in source-to-source transformations and implements several programming models such as OpenMP, OmpSs and StarSs, whereas it is also able to deal with CUDA and OpenCL codes.

We can distinguish two main parts in the compiler. The first one is a fronted with full support for C, C++ and Fortran syntaxes that gathers all the symbolic and typing information into the compiler intermediate representation. The second one is a pipelined sequence of phases that introduces all the source-to-source transformation features and the specific characteristics of the target programming model and runtime.

It provides a plugin-oriented environment that easily allows to be extended, where plugins are represented for one or several phases that can be straightforwardly incorporated to its pipeline and dynamically enabled or disabled when corresponding. This allows Mercurium to manage code restructuring for different target devices just implementing a new phase for each particular architecture.

The compiler also has several *walkers* or tools which help in the management of the source code information through all the compilation phases. It also has a powerful representation of raw source code that allows developers to directly insert, modify and parse source code in any phase of the compilation process.

5.3 Nanos++ Runtime

Nanos++ [56] is an extensible and runtime library developed at BSC mainly aimed at providing support for the OmpSs parallel programming model, although it also supports OpenMP and Chapel[]. It is able to deal with SMP architectures, GPUs and clusters of both GPUs and SMPs.

The most important service of Nanos++ is to manage task parallelism with support for synchronizations based on data-dependencies. In this context, Nanos++ also provides support for efficiently keeping coherence across different address spaces, such as in heterogeneous systems with GPUs, by means of a software directory and cache.

Mercurium compiler is specialized in automatically generating the corresponding calls to the Nanos++ runtime from the user directive and annotations, so programmes do not have to write them by hand.

Proposal: User-directed Vectorization

As described in previous chapters, computer architecture and systems are tending to be heterogeneous where SIMD-alike processing units are coming into prominence. However, compilers are still far from being able to automatically take advantage of their whole computational power, in spite of the fact that the vectorization problem is a well-known topic studied over the years.

In an attempt to improve such compilers auto-vectorization capabilities targeting heterogeneous architectures and accelerators, we propose a generic and portable solution where the programmer can assist the compiler in this process, highlighting what regions of the code are candidates to be vectorized.

6.1 SIMD Directive

As the high point of our proposal, we define a new compiler directive that guides the compiler in the auto-vectorization process, solving the problem of auto-vectorizing codes with characteristics that normally inhibit the compiler from vectorizing.

In the particular implementation of C/C++, we denote it as `#pragma omp simd`. Listing 4 shows a formal description of it.

```

1  #pragma omp simd [(list-item)]
2      structured-block
3
4  #pragma omp simd
5      function-declaration | function-definition

```

Listing 4. SIMD directive specification

The first rule means that the directive may be used on structured blocks, like for-statements, to suggest to the compiler what blocks are potentially *simdizable*. *list-item* denotes a list of symbols that could be included to determine which arrays or pointers from outside of the block should be treated in a vector way. Depending on the compiler

skills (alias analysis, subscript analysis, dependence analysis) *list-item* could be necessary or not.

The second rule introduces the use of the SIMD directive on function declarations/definitions. This kind of annotations is particularly important for vectorizing regions of code with function calls, since they are one of the main reasons why the compiler decides not to vectorize the code. In this case, annotated functions and their parameters will be completely treated in a vector manner so that the *list-item* is not needed.

We use the same structure as OpenMP directives because of the close semantic meaning that our directive has describing *simdization* possibilities with respect to the OpenMP ones have describing Thread- and Task-Level Parallelism. Programmers will be able to use this directive to highlight to the compiler which portions of code are *simdizable*/vectorizable, in the same way that, for example, we can use the corresponding OpenMP directive to define a parallel area.

The most important peculiarity of this new directive is that is completely generic in the sense that programmers do not have to specify any particular characteristic of the target architecture, like the vector width. In this way, from a single high-level source description, the compiler will be able to vectorize the code using the information from the underlying architecture/system, other directives of the corresponding programming model or flags specified at compile time.

```
1  #pragma omp simd
2  float foo(float X)
3  {
4      ...
5  }
6
7  void main(int args, char argv[])
8  {
9      ...
10     float *a = (float *) malloc(N * sizeof(float));
11     float *b = (float *) malloc(N * sizeof(float));
12     float *c = (float *) malloc(N * sizeof(float));
13
14     #pragma omp simd(a, b, c)
15     for (i=0; i < N; i++)
16     {
17         c[i] = a[i] + b[i] + foo(a[i]);
18     }
19     ...
20 }
21
```

Listing 5. SIMD directive usage example

Listing 5 contains an example using the directive on a loop that operates on three vectors and performs a function call. As depicted, our directive is used to suggest the compiler which loops and functions should be *simdized*. Then, the compiler will analyse the annotated code and will determine if it is possible to proceed with the *simdization* or not. In this example, *a*, *b* and *c* are included in the for-statement directive because they are pointers to the input and output arrays that will be accessed in a traditional vector way.

6.2 Implementation Design

As part of our proposal, we also describe the scheme with the high-level compilation phases that a generic compiler should incorporate in its structure so as to properly add the new philosophy of the SIMD directive (Figure 2).

From our point of view, two coarse-grained phases should be incorporated in the target compiler in order to offer a generic and portable solution: a *Generic SIMDization* phase, necessary to transform the code into a generic *Intermediate SIMD* representation, and a target-architecture back-end (one per supported SIMD instruction set or architecture), which turns the intermediate representation into the specific instruction set architecture.

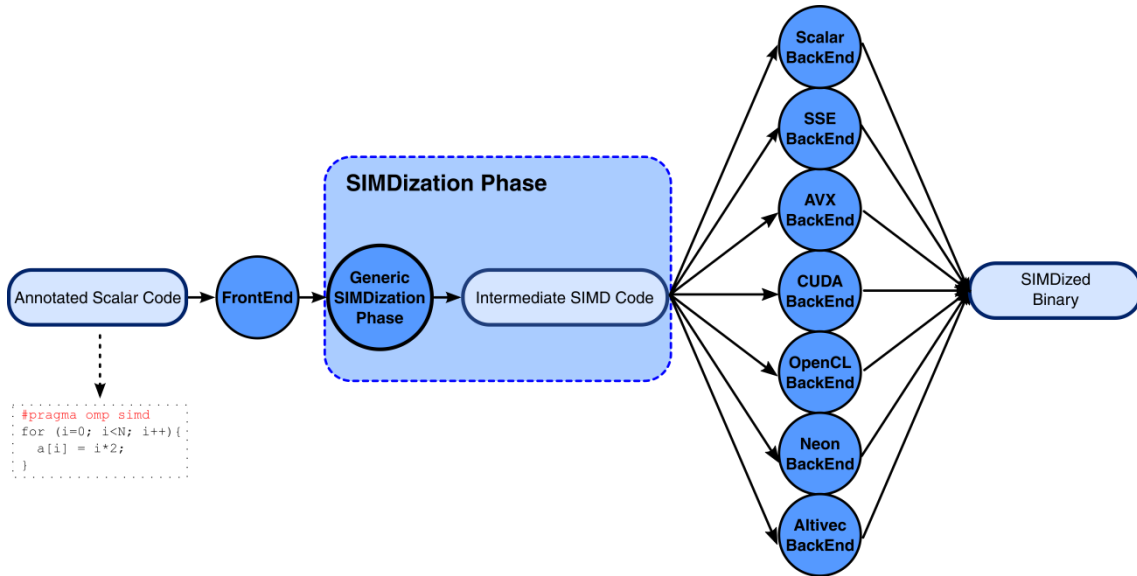


Figure 2. Generic compiler implementation scheme of SIMD directive

In the *Generic SIMDization* phase, once the code has been lexically and syntactically analysed, among other possible things, the compiler will examine the annotated code in order to determine if it is vectorizable or not, with the special consideration of the annotations, which imply that some of the limiting factors in the

auto-vectorization process do not have to be taken into account, such as pointer aliasing and the possible side-effects of the function calls.

Thus, the compiler will *simdize* the code when possible, using a generic representation that allows later specific-architecture phases to properly interpose their particular constraints. This resulting *Intermediate SIMD* code may change among compilers, but it should be a code simdized in a generic way where any specific code generation was reachable from.

At the end, the corresponding target-architecture back-end will translate the intermediate code to the architecture-specific code and will impose its particular constraints, generating the final *simdized* binary.

Design & Implementation

This chapter describes the particular design and implementation decisions of our proposal on the Mercurium C/C++ source-to-source compiler as part of a new feature in the OmpSs parallel programming model.

We limit our implementation to only deal with the SSE 4.1 and previous updates of the SSE instruction set, but our design allows to easily be able to extend the implementation to other SIMD domains.

Only the most interesting details have been described avoiding trivial or very low-level compiler-specific details without interest in this context.

7.1 Design

7.1.1 SIMD Scheme on the Mercurium Compiler

From the point of view of a source-to-source compiler, as in the case of Mercurium, the generic scheme is slightly different to the one depicted in Figure 2. Figure 3 describes the new scheme adapted to our compiler for the particular implementation of SSE code. Only Mercurium SIMD phases are showed for simplicity.

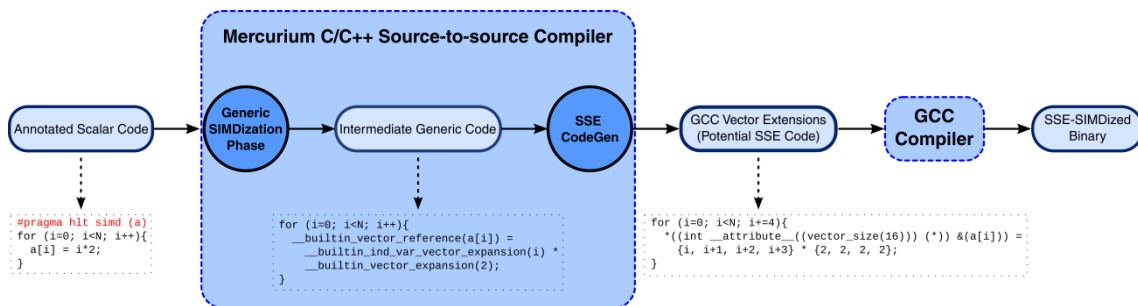


Figure 3. SIMD directive implementation scheme in Mercurium Compiler.

In the *Generic SIMDization* phase, the annotated scalar source code is translated to an intermediate and generic representation. We opted for a generically-*simdized* code using the concept of generic vector described in section 7.1.2. This kind of generic

representation allows the compiler to generate specific code with any kind of constraints in a later step. We decided to incorporate this phase into the existing High-Level Transformations (HLT) one in Mercurium. The HLT phase is responsible for applying high-level transformations on the source code, like loop unrolling and loop interchange among others [58], so it is a very appropriate place in order to add our first SIMD transformation engine.

The *SSE Code Generation* phase is a specific phase that turns the previous intermediate code with generic vectors into a specific SSE representation. It has been built as a new code generation phase following the same structure than others already implemented in the compiler. In the same way, new SIMD code generation phases can extend our implementation to other specific SIMD architectures.

In source-to-source compilers, native compilers impose certain restrictions in the generated source code, because it must be understandable and easy to process by them in order to produce the optimal assembler code that we are looking for.

For this reason, we studied two different possibilities for the Mercurium output SIMD code: generating pure SSE code using low-level intrinsics or generating SIMD code based on GCC vector extensions [37]. Listing 6 contains an example of both approaches.

```
1  __m128i a, b, c, d;
2  d = _mm_sub_epi32(_mm_add_epi32(a, b), c);
3                                     (a)
4
5  int __attribute__((vector_size(16))) a, b, c, d;
6  d = a + b - c;
7                                     (b)
```

Listing 6. SSE code (a) vs. GCC vector extensions (b)

Pure SSE code is much more complicated in terms of implementation because the compiler has to intercept, analyse and process absolutely every operation in the source code individually, substituting each one by a specific intrinsic. The main problem is that, sometimes, there are several intrinsics that may be used for the same operation and then, extra information from proficient analysis techniques is needed to choose the most appropriate. Unfortunately, these techniques are still under development in our compiler.

GCC vector extensions force us to trust GCC to lead the final translation to the specific SSE intrinsics. An important advantage is that this vector extension is architecturally independent so that it would be really simple to change the target architecture in order to generate code for other SIMD extensions.

On the downside, this representation is not supported by the majority of compilers, so not all compilers will be able to perform the back-end duties when this extension is activated.

Despite the disadvantages, we chose GCC vector extension as output code for its facilities in terms of implementation and deployment on different architectures, although we also use SSE intrinsics directly when a very specific functionality is not supported through them.

7.1.2 Generic Vectors

The concept of *generic vector* is very close to that of *virtual vectors* [40]. We define a generic vector as an abstract vector with an arbitrary length and no alignment constraints which means that it is not tied to any specific architecture. These characteristics allow any later architecture-specific transformation with particular constraints.

```

1      int __attribute__((generic_vector)) a;
2      int __attribute__((generic_vector)) b;
3      int __attribute__((generic_vector)) c;

```

Listing 7. Representation of three generic vector variables in C/C++

To represent them, we decided to add a new attribute called *generic_vector* to the compiler semantic analysis, which extends any scalar data type in declarations with its corresponding generic vector connotation, as described in Listing 7.

This abstraction is crucial in the implementation and design of our proposal and leads the *Generic SIMDization* phase becoming a new data type in the Intermediate SIMD Code.

7.1.3 Generic Functions

We may also deal with function declarations/definitions in order to address the *simdization* of source code with function calls. An intermediate and generic representation for those functions called from inside an annotated SIMD region is shown in Figure 4.

Each target function will be represented by an instance of the generic entity *Generic Function*. These entities represent an abstraction of functions without specific implementation details. In turn, each generic function may also have several *Specific Function* instances with the specific representation for different architectures and situations. For example, the same function could have a representation to be executed in an AVX-capable CPU, another vectorized implementation from the scalar version using SSE extensions, a different one using any particular SSE intrinsic that implements the

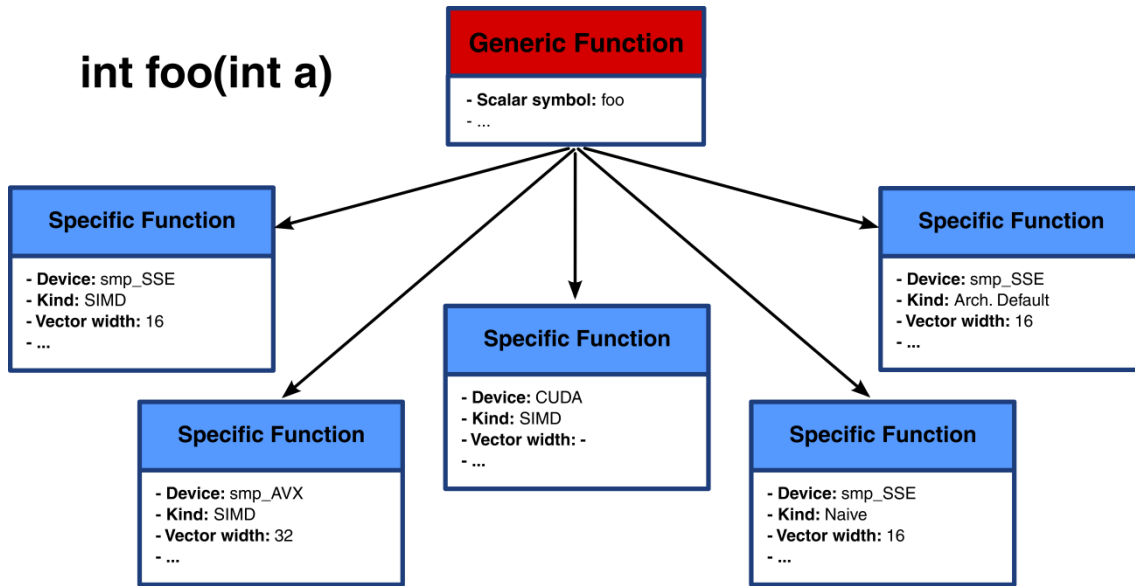


Figure 4. Generic representation of SIMD functions

same functionality and other based on CUDA. From all of these specific representations, the compiler will determine the most appropriate for each case, depending on the nature of the function, the target architecture/system and its characteristics.

Thus, in addition to the specific implementation depending on the target architecture, we define four kinds of specific functions:

- **Naïve:** Trivial SIMD implementation that encapsulates several calls to the original scalar version. This implementation is interesting to continue *simdizing* codes with function calls with not available or not *simdizable* code.
- **SIMD:** Functions annotated with the SIMD directive that have been successfully *simdized*.
- **Architecture default:** Default SIMD functions registered in the compiler, associated to a specific architecture. For example, some transcendental operations have its particular SIMD instruction in some architectures. They may be a simple intrinsic, a non-local function, or a specific function suggested by the compiler.
- **Compiler default:** Functions generated by the compiler to implement any specific functionality, but not directly associated to any scalar function.

In this way, we could have several implementations for different architectures of the same scalar function or even different implementations aimed at the same specific architecture but with different goals.

7.2 Generic SIMDization Phase Implementation

7.2.1 SIMD Directive

The first step to enable our user-directed vectorization approach is to register the new directive in the aforementioned High-Level Transformation phase, using the corresponding tools that Mercurium implements for this purpose.

Once this is done and the compiler recognizes the new directive notation, we endow it with the associated functionality. In first place, when the compiler finds a SIMD directive, the compilation flow diverges in two paths depending on the annotated structure: a for-statement or a function definition.

In the case of a for-statement, the compiler goes through the generic vectorization process and then, it puts a mark on the for-block to recognize it in later phases as a SIMD one and sets some attributes that easily provide information about the induction variable, the original step, the original conditional expression and the shorter data type in the for-statement. Mercurium also generates an epilogue code from the original scalar one that does not contains any SIMD transformation, but just the corresponding modification on the bounds of the loop to compute only the remaining iterations that are not multiple of the vector width.

Function definitions are easier to process. Once the compiler has generically vectorized the function code, it adds the function to the Generic Function infrastructure as a new SIMD one.

7.2.2 Generic Vector Data Type

The generic vector data type implementation, based on its representation as a new attribute that complements the scalar data type, is quite simple. We basically added the corresponding semantic meaning to the new keyword `generic_vector` as part of the `__attribute__` notation.

Regarding the semantic representation, since GCC fixed-length vector types were supported in our compiler, we decided to follow a similar representation for the generic vector types, but using a vector of length zero in the case of generic vectors.

However, the associated operations on this data type required much more work from the point of view of the C/C++ type checking. We carefully added support for the vast majority of assignments, arithmetic, bitwise and comparison operations on generic vector data types, although we mainly focused on the operations needed in our benchmarks.

7.2.3 Generic Vectorization Process

The generic vectorization process may be summarized as replacing scalar data types by its corresponding generic vectors data type. To achieve this purpose, several source code transformations take place in a non-trivial recursive way where several states determine which transformations must be applied in each particular context.

The simple one basically consists of substituting the original type of local variable declaration by its equivalent generic vector data type.

To address scalar constants, function calls, scalar variables or vector load/stores on arrays present in the *list-item*, we need further resources than simply interchange data types to maintain the generic vector consistency. Thus, the compiler makes use of special built-ins that encapsulate those scalar elements in generic vector data types, while they in turn help to highlight those situations that require a special processing in following phases. The most important built-ins are described in Table 1.

Built-in Name	Goal
<code>__builtin_vector_expansion</code>	It encapsulates a scalar constant or loop invariant that needs to be promoted to vector data type.
<code>__builtin_vector_reference</code>	It encapsulates an array access from arrays present in the <i>list-item</i> of the SIMD directive that will become a vector load/store.
<code>__builtin_generic_function</code>	It encapsulates a function call.
<code>__builtin_ind_var_vector_expansion</code>	Similar to <code>__builtin_vector_expansion</code> but this denotes the induction variable promotion to vector when it is used outside an array subscript.
<code>__builtin_vector_conversion</code>	It denotes a conversion between different data types.
<code>__builtin_vector_subscript</code>	It encapsulates an array access subscripted by a vector data type (gather/scatter operation).

Table 1. Description of Generic Vectors Built-ins

These special built-ins have an overloaded data type that is calculated depending on the type of the input parameters. Hence, the same built-in may return different generic vector data types. In addition, these built-ins are allowed to be used as the left side of expressions (l-type).

Once data types and built-ins have been properly incorporated, we get the final version of our intermediate generic code. It is important to note how the boundaries in annotated loop statements cannot be modified yet because the compiler still does not have information about the target architecture.

Some examples of intermediate code are shown in the appendix A.

7.2.4 Generic Functions Infrastructure

In order to implement the generic function functionality described in section 7.2.4, Mercurium makes use of two C++ classes to represent the information of each generic function (class *GenericFunctionInfo*) and its corresponding specific implementations (class *SpecificFunctionInfo*).

GenericFunctionInfo basically stores the symbol of the scalar function and a multi-level multi-map data structure that contains the different specific versions of such a generic one, indexed by the target device and width of the specific function.

SpecificFunctionInfo keeps the name and the kind of the specific function, the target device which has been created for and the width of the SIMD unit, if applicable.

With the aim of hiding the implementation details to the users, another class, *GenericFunctions*, provides the facade pattern, well-known in software engineering design. Thus, this last class plays the role of user interface for the generic function subsystem, offering operations like *add_generic_function* and *add_specific_definition*. It also contains a map with the different function symbols and their associated *GenericFunctionInfo* instance.

7.3 SSE Code Generation Phase Implementation

In the SSE Code Generation Phase, Mercurium generates GCC vector extension (See section 7.1.1) from the Intermediate Generic Code that will be later transformed to the corresponding SSE assembler instructions by the native compiler, GCC in our case.

7.3.1 SSE Vectorization Process

The first step to vectorize the generic code in a SSE way is to introduce the specific constraints associated with this vector extension.

For both, for-statements and function definitions, the generic vector data type represented as `__attribute__((generic_vector))` is replaced by its fixed-size equivalent `__attribute__((vector_size(16)))`, where 16 is the length in bytes of the SSE vector registers.

With the vector length known, for-statement brings information from the Generic SIMDization Phase with the smallest data type that will determine the unrolling factor. Thus, the loop step is now increased accordingly to such unrolling factor. The compiler also adjusts the upper-bound properly when it is not multiple of the unrolling factor: $new_upper_bound = old_upper_bound - (new_step - 1)$, and it studies if the epilogue must be preserved or removed consequently.

It is important to note that we are not performing a traditional unrolling and vector packing in order to vectorize the loop because these techniques are implicitly

applied in the transformation from generic vectors to fixed-length vectors. For this reason, codes that work on mixed data length (See section 7.3.5) require a special treatment so as to replicate those instructions on data types larger than the smallest one that need several vector registers per operand to reach the unrolling factor. Hence, the aforementioned replication is performed on full statements and particular instructions when necessary.

7.3.2 Vector Loads/Stores

Vector loads and stores deserve special mention in this source-to-source transformation process. As described in 7.2.3, the generic representation of vector loads and stores in the intermediate code is conducted by `__builtin_vector_reference`, which encapsulates a scalar array access that must become a vector access.

```

1          hostCallPrice[y]
2          (a)
3
4      *((float __attribute__((vector_size(16))) *) )&(hostCallPrice[y]))
5          (b)
```

Listing 8. Scalar array access (a) and its corresponding vector access (b)

To achieve this purpose, Mercurium generates a slightly extensive code that simply extracts the address of the scalar access, converts such a scalar reference into its vector equivalent and finally performs the access in a vector way. Listing 8 shows an example of a scalar array access and its resulting vector code.

This code is efficiently translated to the specific SSE load instruction and it is easily portable to other SIMD extensions.

7.3.3 Scalar Expansion

Scalar constants, loop invariants and the induction variable were properly encapsulated in the Generic SIMDization Phase through the built-ins `__builtin_vector_expansion` and `__builtin_ind_var_vector_expansion`, respectively. When these two built-ins are found in the current phase, the scalar constant or the induction variable is properly promoted to vector using the GCC vector extension.

```

1          {5.0f, 5.0f, 5.0f, 5.0f}
2          (a)
3
4          {i, i, i, i} + {0, 1, 2, 3}
5          (b)
```

Listing 9. Constant (a) and an induction variable (b) SSE scalar expansion

As depicted in Listing 9, in the case of constants and loop invariants, such as `5.0f` in (a), the constant value or identifier is replicated as many times as needed to fill a fixed-size vector register. In the same way, the compiler extends the induction variable, `i` in (b), but adding the corresponding iteration offset (loop step) in each vector position. This offset value is also adjusted accordingly when the compiler is dealing with replicated statements that require taking into account the previous ones.

7.3.4 Conditional Operator

The conditional operator ‘?’ is also supported in SIMD regions and may be vectorized. We opted for a non-safe implementation for testing, which means that both sides of the condition will be executed in a vector manner and the final result will be selected depending on the condition. For this reason, the programmer must ensure that both sides of the condition do not have side effects if they are executed when not corresponding.

```

1           v = (h == 0) ? v : -v;
2                               (a)
3
4   v = __builtin_ia32_blendvps(-v, v, __builtin_ia32_pcmpeqd128(h, {0, 0, 0, 0}));
5                               (b)

```

Listing 10. SSE implementation (b) of a scalar conditional operation (a)

Listing 10 shows an example of a conditional operation implemented following our approach in SSE. As depicted, both paths of the condition are executed (‘-v’ and ‘v’) and a final scalar selection from both sides is carried out by a ‘blend’ SSE instruction, depending on a vector binary mask. This vector binary mask, and in general any vector comparison operation, is performed by a specific SSE instruction, like the `pcmpeqd128` in the example.

7.3.5 Operations on Mixed Data Lengths and Vector Conversions

One interesting issue that arises when vectorizing codes is how to deal with loops that operate on data types with different sizes. In architectures where vector registers have a fixed size, as SSE, the maximum number of elements that fit in a register may vary depending on the data type size of the scalar element. Following the example in Listing 11, in architectures with 128-bit (16 bytes) vector registers, we could do fit up to four single-precision floating-point (4 bytes) or sixteen unsigned-char (1 byte) scalar elements within a vector.

This situation will cause that the most appropriate unrolling factor needed to vectorize the code, based on the number of elements that fit into a vector register, may be different depending on the target data type.

To address this problem, we must distinguish two slightly different cases: different statements that operate on mixed data types (each statement performs operations on data types with the same size) and mixed data types present in the same statement, as shown in the first and the second loop of Listing 11 respectively.

```
1  float f_array[N];
2  unsigned char c_array[N];
3
4  for (i=0; i<N; i++)
5  {
6      f_array[i] = 0.0f;           // float = float
7      c_array[i] = (unsigned char) 100; // uchar = uchar
8  }
9
10 for (i=0; i<N/2; i++)
11 {
12     c_array[i] = f_array[i];     // uchar = float
13 }
```

Listing 11. Two simple vectorizable loops that operates on mixed data lengths

The simplest option to solve both cases is to define the unrolling factor from the number of scalar elements of the largest data type that fits into a vector register, and then apply straightforward conversions between the different types in the second case, as defined in *The C Standard* [54]: to perform operations on different data types, the smaller data type are normally converted to the largest one and the resulting type of the operation will be also the type of this last one.

In our example with single floating-point and unsigned-char data types, the first type would impose an unroll factor of four, so the vectorized code would work with 4-element unsigned-char vectors, wasting the remaining twelve elements. Hence, only four unsigned-char elements would be processed at a time, even when all the operands of a particular operation have unsigned-char type, with the resulting loss of performance.

In our solution, we opted for a more complex approach that fixes the unrolling factor to the maximum number of elements of the smallest data type that fits into the vector register. This means that those operations with the same data type size as the smallest one will result in one vector operation whilst several independent vector instructions will be necessary for larger data type operations. When conversions from smaller to larger types are required, the corresponding part of the smaller vector data type will be extracted and properly converted to the larger one in each of the several *replicated* vector instances on it. The other way around, when conversions from larger to smaller data types need to be performed, several of the *replicated* vector instances will be converted and packed in a single register.

Going back to our example, the unsigned-char data type enforces an unrolling factor of 16. Therefore, independent floating-point vector operations will be replicated four times to reach the aforementioned number, as depicted in the first loop of Listing 12. When conversion from floating-point data types to unsigned-char data types is required, four floating-point vectors are converted to unsigned-char and packed in only one register, as shown in the second loop.

```
1  float f_array[N];
2  unsigned char c_array[N];
3
4  for (i=0; i<N/16; i+=16)
5  {
6      f_array[i:i+3]      = {0.0f, 0.0f, 0.0f, 0.0f};
7      f_array[i+4:i+7]    = {0.0f, 0.0f, 0.0f, 0.0f};
8      f_array[i+8:i+11]   = {0.0f, 0.0f, 0.0f, 0.0f};
9      f_array[i+12:i+15]  = {0.0f, 0.0f, 0.0f, 0.0f};
10
11     c_array[i:i+15] = {(unsigned char) 100, ..., (unsigned char) 100}; //16 times
12 }
13
14 for (i=0; i<N/32; i+=16)
15 {
16     c_array[i:i+15] = conver_to_char_vector(f_array[i:i+3],
17                                             f_array[i+4:i+7],
18                                             f_array[i+8:i+11],
19                                             f_array[i+12:i+15]);
20 }
21
```

Listing 12. Pseudo-code from vectorizing loops in Listing 11. (128-bit vectors)

The implementation of conversions between different vector data types has been limited for simplicity to those conversions present in our benchmarks (See section 8.4). Thus, the vast majority of conversions between data types with the same size are supported, and only some conversions from larger data types to smaller data types were included. Conversions from smaller data types to larger data types have not been implemented yet.

When the data types have the same size, like `int` and `float`, the conversion is normally carried out by a single SSE instruction, like `cvtdq2ps`.

When this conversion is between types of different sizes, a more sophisticated solution needs to be performed. In addition to packing operations described previously, sometimes it is not possible to convert a data type directly to a different one, as in case of vector conversions from `float` to `unsigned char`, where `float` vectors must be converted to `int`, then packed and converted to `short int` and finally packed and converted to `unsigned char`. Listing 13 shows exactly how this last conversion is

implemented in our compiler using SSE packing (*packusdw128*, *packuswb128*) and conversion (*cvttps2dq*) instructions.

```
1  static inline unsigned char __attribute__((vector_size(16)))
2  __conv_float_to_uchar_smp16(
3  float __attribute__((vector_size(16))) vf0,
4  float __attribute__((vector_size(16))) vf1,
5  float __attribute__((vector_size(16))) vf2,
6  float __attribute__((vector_size(16))) vf3)
7  {
8      int __attribute__((vector_size(16))) vi0, vi1;
9      short int __attribute__((vector_size(16))) vs0, vs1;
10     vi0 = __builtin_ia32_cvttps2dq(vf0);
11     vi1 = __builtin_ia32_cvttps2dq(vf1);
12     vs0 = __builtin_ia32_packusdw128(vi0, vi1);
13     vi0 = __builtin_ia32_cvttps2dq(vf2);
14     vi1 = __builtin_ia32_cvttps2dq(vf3);
15     vs1 = __builtin_ia32_packusdw128(vi0, vi1);
16     return (unsigned char __attribute__((vector_size(16))))
17     __builtin_ia32_packuswb128(vs0, vs1);
18 }
```

Listing 13. SSE Single-precision floating-point to unsigned-char vector conversion

7.3.6 Gather Operations

Our implementation includes support for a special sort of gather operations that arise when some serial codes are vectorized. This is the case of arrays that are treated as vectors and are indexed by a variable local to the SIMD region. When this variable becomes a vector with non-stride-one values, it is not possible to perform a simple vector load so a gather operation is needed.

```
1  static inline int __attribute__((vector_size(16)))
2  __vector_subscript ( int subscripted[],
3                      int __attribute__((vector_size(16))) subscript)
4  {
5      int __attribute__((vector_size(16))) result =
6      {subscripted[ __builtin_ia32_vec_ext_v4si(subscript, 0)],
7      subscripted[ __builtin_ia32_vec_ext_v4si(subscript, 1)],
8      subscripted[ __builtin_ia32_vec_ext_v4si(subscript, 2)],
9      subscripted[ __builtin_ia32_vec_ext_v4si(subscript, 3)]};
10     return result;
11 }
```

Listing 14. Gather operation using SSE

Since SSE does not provide an efficient gather instruction, this functionality is implemented in a scalar manner extracting the different values from the vector index

and building a vector register from the scalar values of the array indexed element by element, as described in Listing 14.

This implementation allows the compiler to continue vectorizing codes that contains this non-traditional and directly unsupported vector operation.

A more general implementation of these kinds of operations will be soon available. We are waiting to the subscript analysis technology, now under development in our compiler.

7.3.7 Architecture and Compiler Default Functions. ACML Support.

Many compiler-default functions were included in our Generic Functions infrastructure in order to implement some functionality, like vector conversions and gather operations, for the SSE specific instruction set.

Some other architecture-default functions were hand-coded and added to offer a specific-architecture vector alternative to some scalar functions, like *abs*, *fabs*, *sqrtd*, *sqrt*, *floorf* or *ceilf* math functions, among others.

However, the specific-architecture vector functions from the AMD ACML vector math library [39] were the most significant and powerful incorporation to our functions subsystem. Nowadays, Mercurium is able to automatically emit calls to the most important transcendental math functions, like single-precision and double-precision versions of *exp*, *log*, *log2*, *log10*, *pow*, *sin* and *cos*. This set of functions is only enabled when the flag `--acml` is used at compile time.

7.4 Limitations of the current implementation

As expected, our implementation has several limitations in order to vectorize certain codes. For example, Mercurium is currently not able to deal with *structs* or classes, scalar reduction operations, loops or functions with nested loops or if-statements and non-stride-one memory accesses. Additionally, only some conversions between certain types are supported and the implementation of the conditional operator is non-safe, which means that if there are operations with side-effects in any of the two paths of the condition, the behaviour of the program is unknown and the result could be not right.

The compiler analysis is very limited in Mercurium. This means that the compiler cannot determine some cases in which the annotated code could not be *simdizable*. This could result in a wrongly vectorized code that could or not pass the native compilation phase. For this reason, we are also working on compiler analysis at the source-to-source level to improve the knowledge of the compiler in this regard and be able to perform more advanced and appropriate transformations in the presence of some kind of dependencies and/or code patterns.

7.5 CUDA and OpenCL approaches

In an early state of the project we implemented minimal support for automatically generating CUDA and OpenCL kernels just as a proof of concept. Such implementations are not updated to the latest features of the current version and for that reason they will not be included in this project, but the implementation of the prototypes demonstrates the genericity of our approach and how from the same intermediate code it is possible to get specific code for such a different architectures and programming models.

Experimental Results

8.1 Environment

To evaluate the performance of our vectorization proposal, we used a system with the characteristics listed in Table 2. As described, each core is SSE-capable up to the 4.1 version.

Multiprocessor Family	Intel Xeon E7450, x86_64
#Cores per multiprocessor	6 cores
Frequency	2.4 GHz
L1 cache	32 Kbytes per core
L2 cache	2 Mbytes shared per core pair
L3 caches	12 Mbytes shared
SSE Capability	SSE, SSE2, SSE3, SSSE3 and SSE4.1
#Multiprocessors (#Total Cores) in the system	4 multiprocessors (24 total cores)
RAM	48 Gbytes

Table 2. System hardware summary

Regarding software, Table 2 summarizes some information about the operating system and compilers version. We used the latest Intel C/C++ Compiler [36] and Intel [59] and AMD [60] OpenCL SDKs production versions. GCC compiler 4.5.3 [62] was selected because it provides better stability than the newest 4.6.0 release.

Operating system	SUSE Linux Enterprise Server 11 (x86_64)
Kernel version	Linux 2.6.32.12-0.7-default SMP x86_64 GNU/Linux
Gnu C/C++ compiler	4.5.3
Intel C/C++ compiler	12.0
Intel OpenCL SDK	1.1
AMD APP OpenCL SDK	2.4

Table 3. System software summary

8.2 Definition and Methodology of the Experiments

We mainly focused our experiments on two of the most important C/C++ production compilers, GNU C/C++ Compiler and Intel C++ Compiler, with intent to evaluate the auto-vectorization capability of both and comparing their results against our user-directed vectorizer in Mercurium. We also included additional experiments for the SIMD directive implemented in the Intel C/C++ Compiler (See section 2.3.4).

Additionally, we evaluated OpenCL versions of some of the benchmarks that were available in the AMD [60] and IBM [63] SDKs. We also implemented our own hand-coded OpenCL and SSE versions in a few benchmarks.

Looking for a comparison as fair as possible, we adjusted the compilation flags to obtain results with the same precision among all different compilation environments. Table 4 shows the most important flags combination.

GCC Scalar	-O3 -lm -fno-tree-vectorize
GCC Auto-vect	-O3 -lm -msse4.1
GCC Auto-vect + ACML	-O3 -lm -msse4.1 -mveclibabi=acml
GCC SSE	-O3 -lm -msse4.1
ICC Scalar	-O3 -no-vec -fp-model precise
ICC Auto-vect/SIMD	-O3 -msse4.1 -fp-model precise
ICC Auto-vect/SIMD + MKL	-O3 -msse4.1 -fp-model precise -mkl=sequential
ICC SSE	-O3 -msse4.1 -fp-model precise
Mercurium Scalar	-O3 --ompss -lm --Wn,-fno-tree-vectorize
Mercurium SIMD	-O3 --simd -lm -msse4.1
Mercurium SIMD + ACML	-O3 --simd -lm -msse4.1 --acml
Mercurium SSE	-O3 --ompss -lm -msse4.1
OpenCL (Intel & AMD) [GCC]	-O3 -lOpenCL

Table 4. Compilation flags of each particular configuration and compiler

We tested several vector math libraries in some benchmarks with transcendental math function calls. In the case of GCC and Mercurium, we chose the ACML library [39]. ICC already includes a vector math library by default (SVML [64], Short Vector Library Math) although we tested the Intel MKL [38] (Math Kernel Library) as well. ICC was also configured with the (non-vector) GCC GNU Math Library [65] in order to get a comparable ICC performance against GCC and Mercurium when they are using the same math library.

As mentioned in section 7.1.1, we used GCC as Mercurium compiler back-end. For this reason, results between Mercurium and Intel Compiler should be taken with a pinch of salt, due to the notorious differences in optimization techniques that can be applied by them in the presence of the same source code. Those optimizations may

cause very disparate speed-up values because of the generation of a better code but not as result of a better vectorization.

The final execution time speed-up of each experiment is the result of the arithmetic mean of 100 executions on the system described in Table 2, using a single core. The baseline is the execution time of the GCC scalar version with the flags described in Table 4. It is important to note that only the execution part of the algorithm is in the timed region, leaving out of the time the remaining phases, like memory allocation/de-allocation. This is especially favourable to OpenCL, where neither runtime compilation nor memory transfers have been taking into account.

8.3 Benchmarks

Seven different benchmarks form the test set: Saxpy/Daxpy, Vector-vector Multiplication, Matrix-matrix Multiplication, H264, Fast Walsh Transform, Blackscholes and Perlin Noise.

Some of them are simple, like Saxpy/Daxpy, Vector-vector Multiplication and Matrix-matrix Multiplication, with the intention of promoting and evaluating compilers automatic vectorization. The remaining benchmarks are more sophisticated, also vectorizable, but not as easy as the first three, especially Blackscholes and Perlin Noise.

Saxpy (Single-precision real Alpha X Plus Y) and its double-precision version, **Daxpy**, are well-known lineal algebra operations included in the most famous algebra libraries. They basically implement a combination of a scalar multiplication and a vector addition. For our tests, we have available a scalar and a hand-coded SSE version with an input/output arrays of 80,000 elements. The problem is computed 180,000 times per execution in Saxpy, and 90,000 times in Daxpy.

Vector-vector Multiplication and **Matrix-matrix Multiplication** are two benchmarks that implement the traditional multiplication of two vectors, element by element (useful for the dot product), and two matrixes, respectively. The loops order in the Matrix-matrix Multiplication was properly interchanged to get stride-one accesses in the innermost loop. In these benchmarks we have available the scalar version of the code with input/output arrays of 400,000 elements each one and matrixes of 1,536x1,536 elements, respectively. Moreover, the algorithm is computed 10,000 times per single execution in the Vector-vector Multiplication and just once in the Matrix-matrix Multiplication benchmark.

H264 is part of the code of commonly used H264 standard for video compression that consists of several loops with high computational density on single-precision floating-point data types, where not all follow a stride-one access pattern. We have the scalar version of the algorithm which process 3,000,000 blocks per execution.

Fast Walsh Transform is a benchmark that comes from the NVIDIA SDK [61] and it performs a dyadic convolution, which is a transformation based on the Fourier ones with applicability to electrical engineering and numeric theory. This benchmark is tested in its single- and double-precision version using 67,108,864 elements as input data. Only the scalar version is available.

The **Blackscholes** benchmark implements a version of the Black-Scholes mathematical model used in the financial market for pricing European-style options. We use a scalar and an OpenCL version taken from the AMD OpenCL SDK. The benchmark is tested with an input data size of 16,000x16,000 single-precision floating-point elements.

Perlin Noise is an image filter used to increase the realism in computer graphics, for example in games. The implementation that we use came from the IBM OpenCL SDK and the computation of the image is performed using single-precision floating-point elements while the image is stored using unsigned-char data types. We use an image size of 10,240x10,240 pixels.

8.4 Results

8.4.1 Saxpy/Daxpy

The simplicity of these two benchmarks lets us to evaluate how far we could improve performance in codes using the compilers auto-vectorization capabilities when they do not inhibit themselves from auto-vectorization.

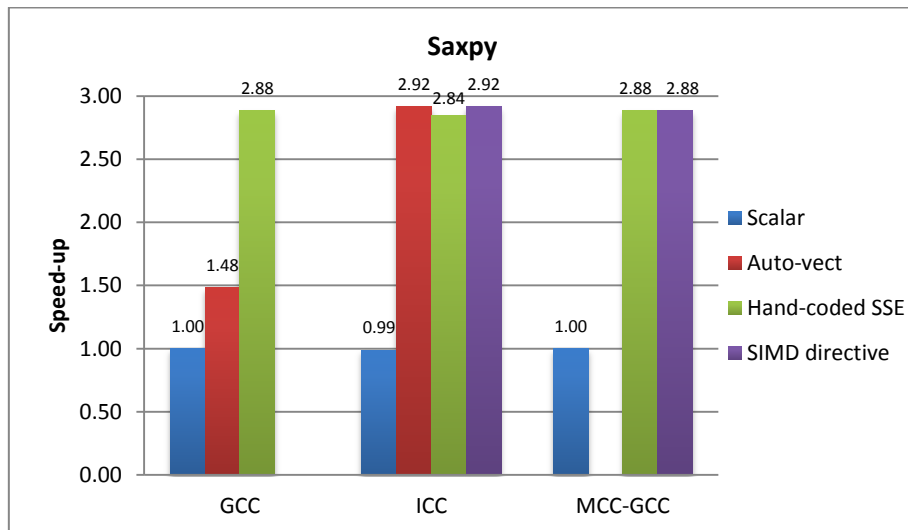


Figure 5. Saxpy speed-up

Figure 5 and Figure 6 show the achieved speed-up using the GNU compiler (GCC), the Intel compiler (ICC) and the Mercurium compiler with GCC as back-end (MCC-GCC). As depicted, GCC is only able to reach a speed-up of 1.48 in Saxpy when auto-vectorization is enabled, which is a very low improvement if we compare it with the 2.88 factor of the hand-coded SSE implementation. On Daxpy, GCC gets a slightly worse performance than the serial version when auto-vectorizing because the larger data size does not compensate the overhead introduced, among other things, by the extra code aimed at memory references analysis.

The ICC auto-vectorization capabilities seem to be better than the GCC ones in this case. Intel user-mandated approach (SIMD) reaches also a similar speed-up, which means that auto-vectorization is doing its best. They both even overcome the speed-up of the hand-coded SSE version.

It is important to note that the ICC better performance is also due to loop unrolling carried out in a factor of four and eight on Saxpy and Daxpy respectively, whilst GCC does not apply this optimization at all. In the Intel hand-coded SSE version, this optimization is also absent, which justifies the lower performance of this particular version with respect to the other two.

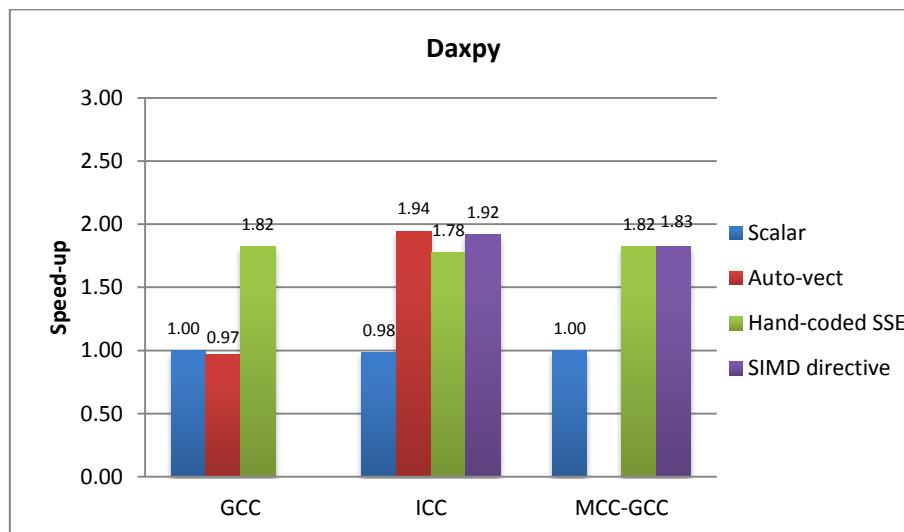


Figure 6. Daxpy speed-up

Our user-directed vectorizer in Mercurium, also without applying loop unrolling optimizations, gets a really good result, improving Saxpy and Daxpy performance as much as the hand-coded SSE approach compiled with GCC (speed-up of 1.83), very close to the best ICC achievements.

8.4.2 Vector-vector Multiplication

In the same way as the Saxpy/Daxpy benchmarks, Vector-vector Multiplication is a simple algorithm that should allow the compiler to get significant improvements with auto-vectorization.

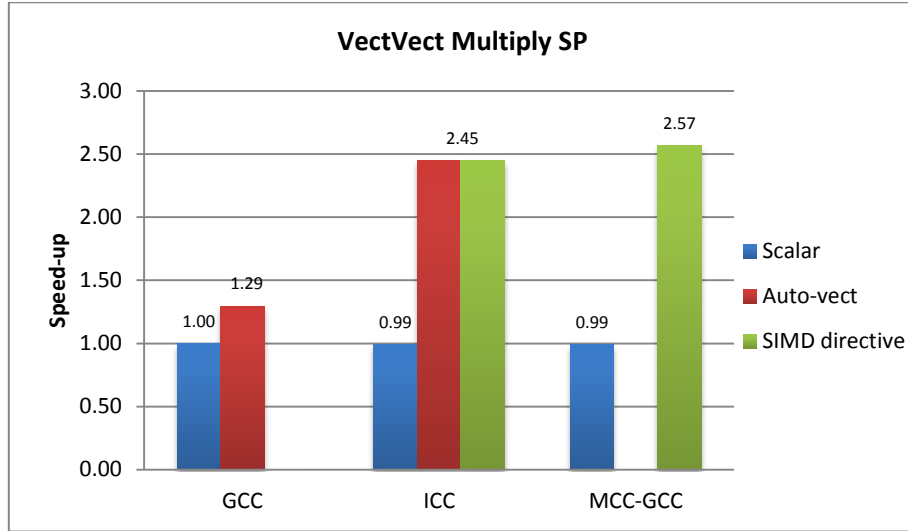


Figure 7. Vector-vector Multiply single-precision speed-up

As shown in Figure 7, the Intel compiler auto-vectorization is able to reach a significant 2.45 speed-up factor, which is the same value achieved by its user-directed vectorization approach. These figures highlight again the good performance of the ICC auto-vectorization capabilities although loop unrolling optimizations are also present in both versions with a factor of four.

In the case of our SIMD proposal on Mercurium, it outperforms even the best ICC speed-up, reaching a factor of 2.57 with GCC as back-end, where unrolling optimizations are again not applied.

Regarding the GNU compiler, it only offers a speed-up factor of 1.29 over the scalar version, which is again a bit far from the performance reached by the user-directed versions in both Intel and Mercurium compilers. Once again, the unrolling optimization is not used in this case.

Analysing the generated assembler code, we can confirm that Mercurium approach yields a better code where tens of instructions for memory address analysis have been removed with respect to GCC and ICC versions, which demonstrates the importance of the Mercurium user-directed vectorization assisting the native auto-vectorization process.

Regarding the OpenCL versions, Figure 8 shows the speed-up achieved by AMD and Intel OpenCL SDK against the Mercurium user-directed approach. As

depicted, our SIMD approach slightly beats them, which confirms that we are generating code as efficient as a hand-coded version.

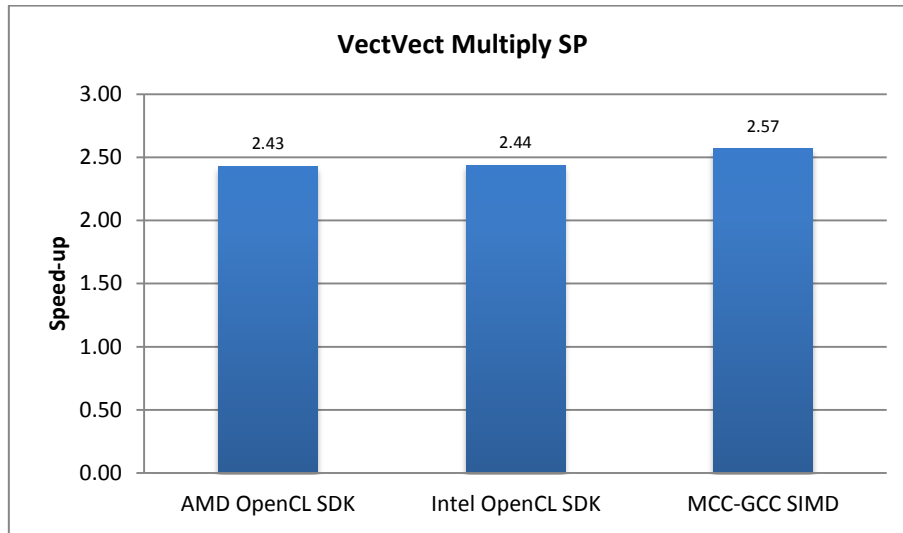


Figure 8. Vector-vector Multiply OpenCL single-precision speed-up

8.4.3 Matrix-matrix Multiplication

Matrix-matrix Multiplication works on two-dimensional arrays, complicating a bit more the vectorization process. This benchmark is interesting to test our vectorization capabilities on multidimensional arrays.

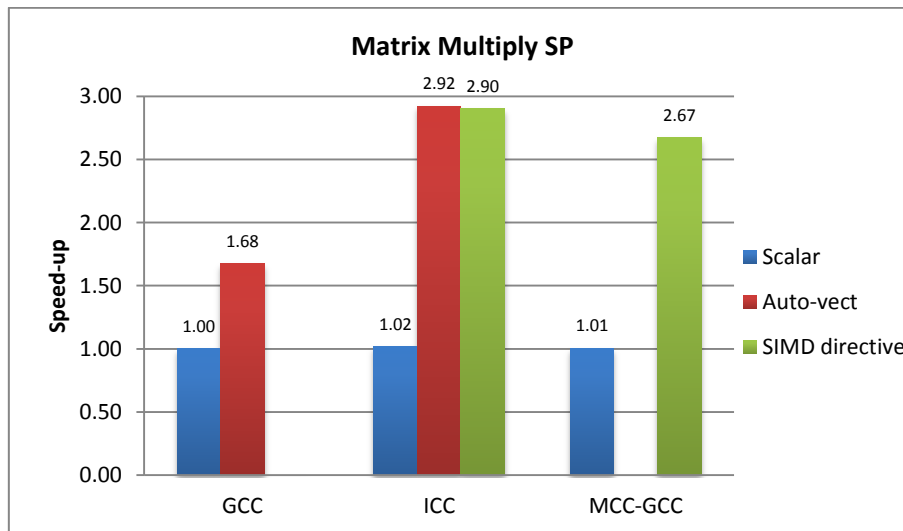


Figure 9. Matrix-matrix Multiplication single-precision speed-up

In this case, we observe (Figure 9 and Figure 10) a similar behaviour to the Saxpy/Daxpy benchmarks. GCC auto-vectorization improves execution performance in a factor of 1.68 over the scalar version, but still far from the ICC automatic and user-guided vectorization improvements, which are 2.92 and 2.90 respectively, in part also

due to the aggressive unrolling levels (four in single-precision and eight in double-precision).

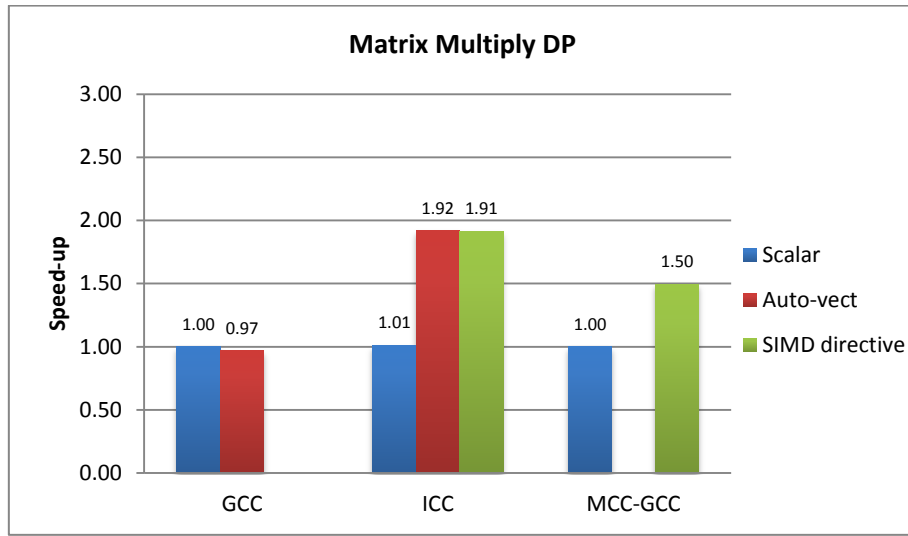


Figure 10. Matrix-matrix Multiplication double-precision speed-up

Meanwhile, Mercurium user-directed vectorization achieves speed-ups of 2.67 in single-precision and 1.50 in double-precision without applying unrolling, which are much better than the reached using GCC auto-vectorization, and close to the better-optimized ICC speed-ups.

8.4.4 H264

Several loops in H264 present a vectorizable structure except one of them that contains non stride-one memory access patterns. From the point of view of the three compilers, this loop is not simdizable, even when the SIMD directive is specified in ICC.

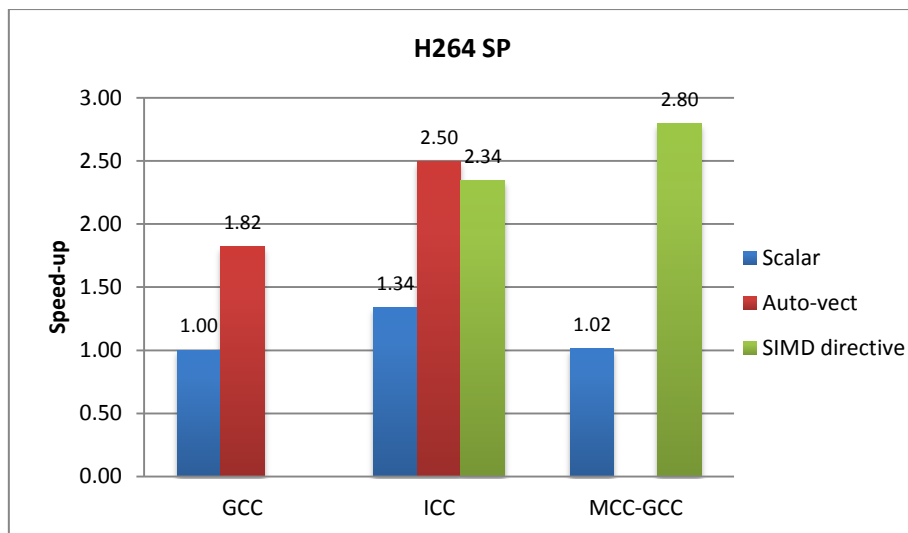


Figure 11. H264 single-precision speed-up

Figure 11 illustrates the different configuration speed-ups. GCC auto-vectorization offers an important 1.82 speed-up factor over the scalar version, followed by the 2.50 and 2.34 speed-ups from the ICC auto-vectorization and user-directed vectorization. The significant difference between the two Intel approaches is due to a loop fusion that takes place in the first one and not in the second one, apparently because of the SIMD annotation of the loops.

Mercurium speed-up overcomes the best GCC and Intel configurations reaching a factor of 2.80. In this case, it is important to realize that the GNU compiler carries out an aggressive loop unrolling optimization in two loops, level six and sixteen respectively, whereas the Intel compiler only unrolls at level two and sixteen. Mercurium also generates better memory operations with respect to GCC which explains the important difference between these two versions.

8.4.5 Fast Walsh Transform

Fast Walsh Transform is the most challenging benchmark for auto-vectorization so far because it is characterized for a peculiar vectorizable structure consisting of three nested loops where the number of iterations in the first one decreases logarithmically, the second one goes upward but the step depends on the previous loop induction variable, and the third one increases its step in a stride-one way with the first loop induction variable as upper bound. Listing 15 shows the structure of the described nested loops.

```

6   for(stride = N / 2; stride >= 1; stride >>= 1){
7       for(base = 0; base < N; base += 2 * stride){
8           for(j = 0; j < stride; j++){
9               ...
10              float T2 = h_Output[base + j + stride];
11              ...
12          }
13      }
14  }
```

Listing 15. Fast Walsh Transform kernel

It is interesting to realize how in the last iteration of the first loop, the innermost loop will perform only two scalar iterations, which means that the last iteration of such loop will not be vectorizable in a 4-way (single-precision) as the previous ones are, although it will be in the double-precision version.

Single-precision and double-precision performance results, showed in Figure 12 and Figure 13 respectively, confirm how the GNU compiler cannot vectorize this benchmark at all supposedly because of the complex structure of the loops. For its part, the Intel compiler is able to get a speed-up of 1.79 and 1.44, single- and double-precision, both in its auto-vectorization and user-directed versions, which once again

shows the great capabilities of the ICC auto-vectorizer. Both single- and double-precision versions have an unrolling factor of eight and sixteen.

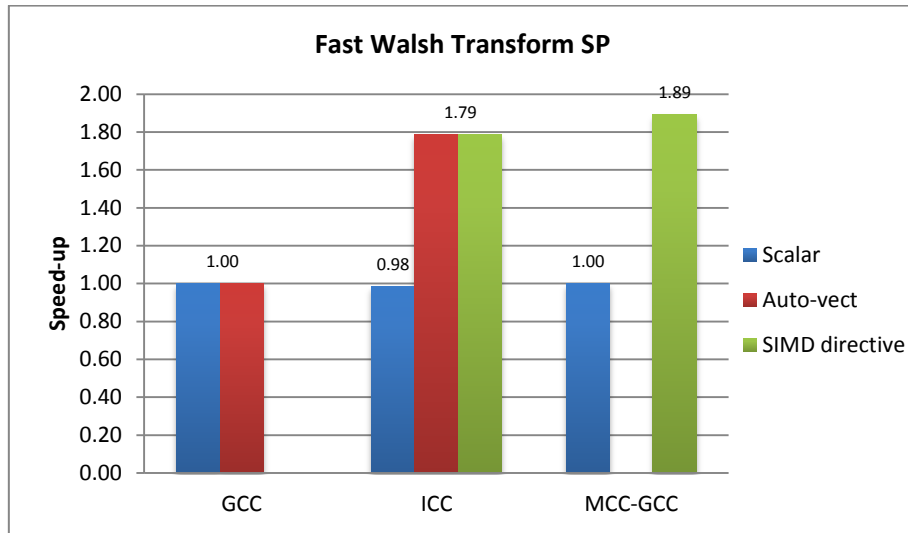


Figure 12. Fast Walsh Transform single-precision speed-up

Meanwhile, our user-directed vectorization reaches a remarkable 1.89 and 1.53 speed-up in the two implementations of the algorithm, where GCC as back-end provides only an unroll factor of two in both.

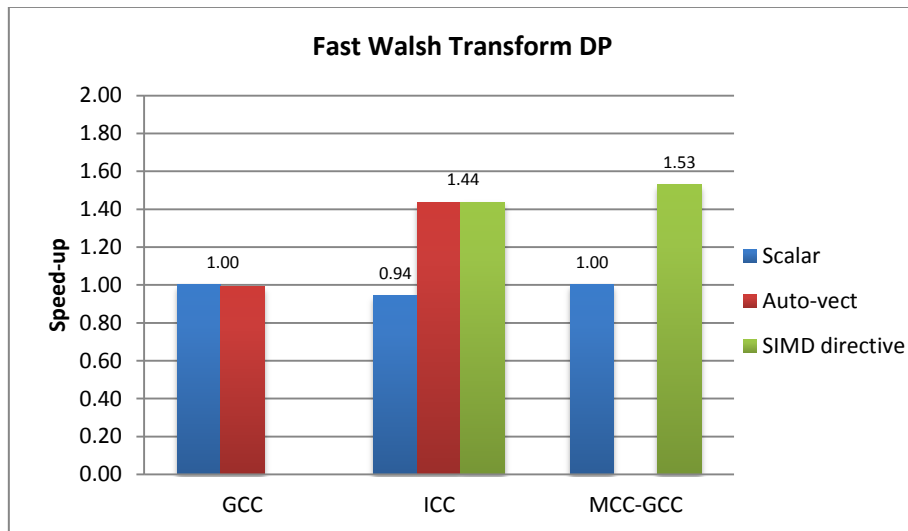


Figure 13. Fast Walsh Transform double-precision speed-up

8.4.6 Blackscholes

The Blackscholes benchmark is not as *simple* as the previous ones seen so far. It has two nested loops with a considerable code length and high computational density with function calls. We find several transcendental functions like square root,

exponentials and logarithms, so it is a good benchmark to test the compilers automatic support for vector math libraries, ACML in the case of Mercurium.

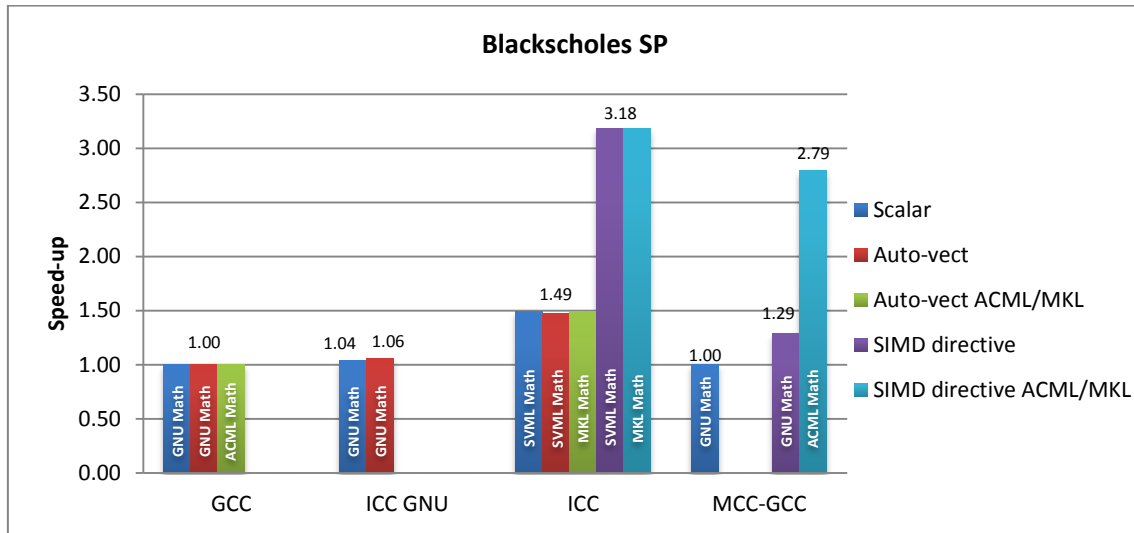


Figure 14. Blackscholes single-precision speed-up

Figure 14 shows the different speed-ups achieved by the different compilers and configurations. As depicted, GCC compiler inhibits itself from auto-vectorizing, even when the ACML math library is activated. Intel compiler auto-vectorization is also disabled either with the GNU (ICC GNU), the SVML and the MKL (ICC) math libraries, although SVML and MKL libraries provide a better speed-up (1.49) than GNU one. However, when the code is annotated with the ICC SIMD directive, we reach an incredible 3.18 speed-up factor over the GCC scalar version, both with the SVML and the MKL Intel math libraries, which reveals that the user guidance is indispensable

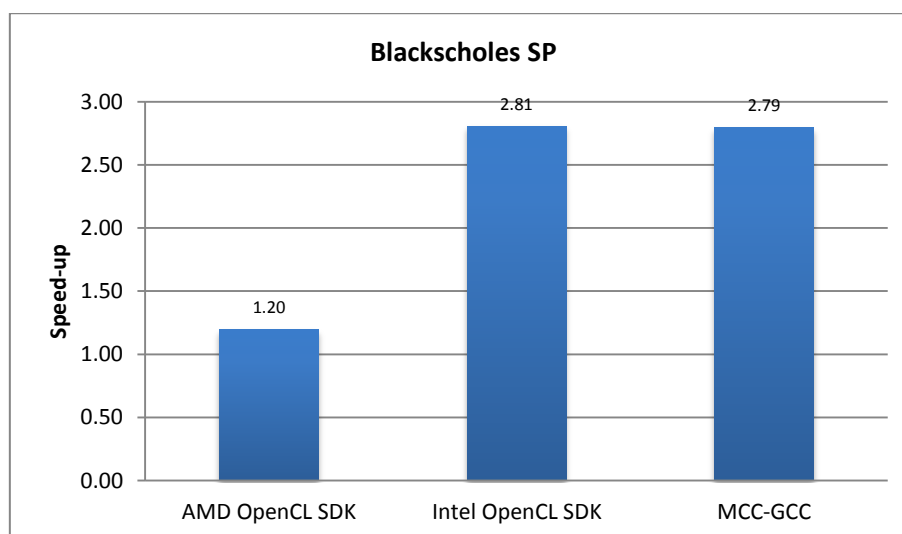


Figure 15. Blackscholes OpenCL single-precision speed-up

to auto-vectorize this code. We were not able to use the SIMD directive in conjunction with the GNU math library.

In our implementation, we improve the execution performance in a 2.79 factor, which is a very impressive number taking into a count that we are using GCC as back-end and a math library better optimized for AMD architectures as opposed to the SVML and MKL ones.

Regarding OpenCL implementations (Figure 15), our result is as good as the one reached by the Intel OpenCL SDK while the AMD one only gets a 1.20 speed-up factor possibly because we are using an Intel architecture.

8.4.7 Perlin Noise

Perlin Noise is by far the most complicated benchmark included in this evaluation. The vectorizable region contains data types with different sizes (`unsigned char` and `float`), conditional operators (`'?'`), nested function calls, non-stride-one vector loads operations (`gather`), and rounding functions.

The data structures of the algorithm were properly simplified in all the configurations to promote the auto-vectorization and minimize the implementation and support for less important features in our compiler.

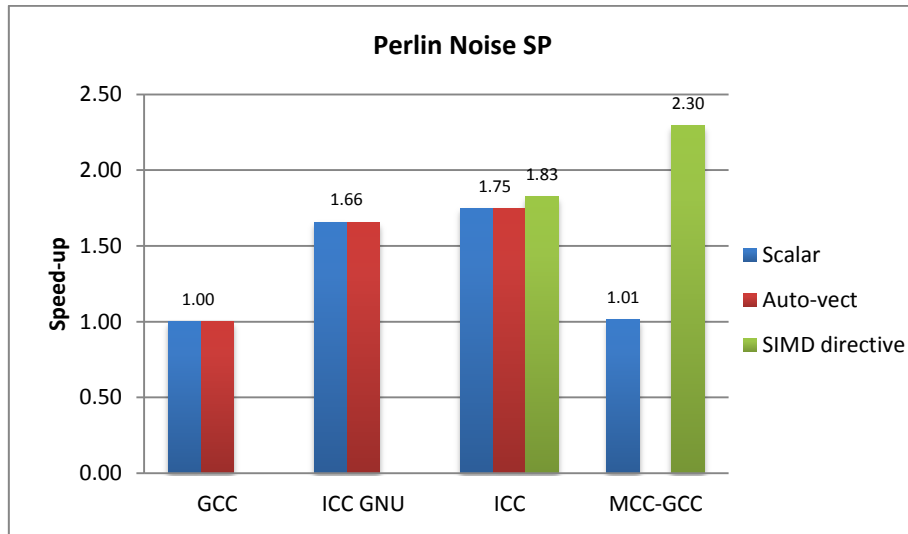


Figure 16. Perlin Noise single-precision speed-up

As depicted in Figure 16, neither the GNU compiler nor the Intel compiler are able to vectorize the code, although the ICC scalar version gets an important improvement over the GCC one which is, among other things, due to the faster implementation of the rounding functions in the default ICC math library (SVML) with regard to the GNU one. The ICC *simd*-annotated configuration slightly increases the speed-up over the auto-vectorization version because of different decisions in the

generation of the code but not due to vectorization since the compiler informs that it is not able to vectorize the code.

The user-directed version in our Mercurium compiler achieves an impressive 2.30 speed-up over the scalar version of GCC, being able to vectorize the code even in presence of conditional and the gather vector load operations.

Comparing the Mercurium achievement against the hand-coded OpenCL implementations (Figure 17), we can observe how the Mercurium speed-up is still better than the 2.23 and 1.27 speed-up factors reached by the Intel and AMD OpenCL SDK, respectively.

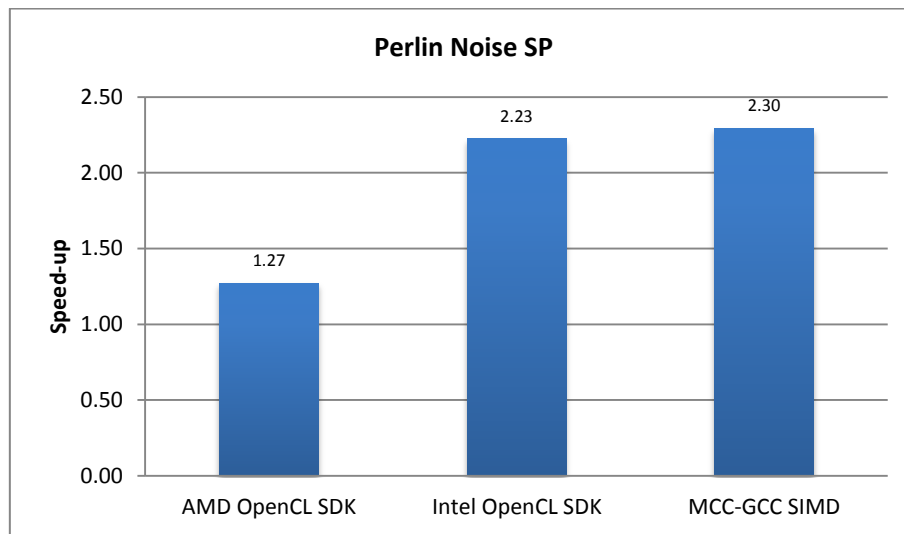


Figure 17. Perlin Noise OpenCL single-precision speed-up

8.5 Discussion

Summarizing the achieved speed-up results, we can confirm how the GNU compiler has important problems in the auto-vectorization process when pointers and function calls are present. The Intel compiler shows a much better auto-vectorization technology in the sense that is able to efficiently auto-vectorize source codes that GCC does not *dare* to vectorize.

However, not even ICC auto-vectorizer is able to vectorize benchmarks such as Blackscholes and Perlin Noise without the assistance of the user, which shows the importance of the user-directed vectorization.

Intel user-mandated vectorization gets significant improvements over the scalar versions, allowing to vectorize codes that the simple auto-vectorization is not able to deal with. However, this approach gets lower speed-ups than auto-vectorization in some cases because the SIMD directive inhibits some compiler optimizations and source code transformations that are performed when the directive is not present. Moreover, despite

the fact that some loops and functions are annotated with such directive, Intel compiler informs that it is not able to vectorize the code.

With regards to our user-directed vectorizer in Mercurium, we can confirm that it is able to vectorize source code much better than the GCC auto-vectorizer, reaching also the same or higher speed-ups than some hand-coded SSE and OpenCL versions, although the AMD OpenCL SDK seems not to be offering its best performance results because of the Intel's underlying architecture. It is also important to realise the significant performance improvements achieved just activating the support for the ACML library math with the corresponding compilation flag.

Comparing our results against the Intel user-directed vectorization it is not entirely fair from the point of view that we are using the GNU compiler as back-end. However, we could see how ICC reaches slightly higher levels of speed-up in those cases where it applies more aggressive optimization techniques, whereas the speed-up is slightly worse when the level of optimization is the same or unfavourable to ICC. Nevertheless, Intel compiler is not able to vectorize the Perlin Noise benchmark, even with its user-guided approach, which is a plus for our user-directed technology.

Conclusions and Future Work

9.1 Conclusion

We conclude this project highly motivated and with high expectations on our user-directed vectorization proposal as part of a combined solution to deal with heterogeneous architectures and systems where SIMD processing units or independent devices are largely responsible for the execution performance.

Although our proposal needs to be extended and refined, we think we have the foundations for developing a solid solution that provides an efficient, portable and generic programming model extension that could be incorporated as part of future OpenMP standard releases.

This project demonstrates that the high-level architecture-independent SIMD directive provides an abstraction layer that allows programmers to easily guide the compiler in the vectorization process. The generic intermediate representation allows a generic *simdization* of the source code without particular architecture-dependent constraints and characteristics, which can be efficiently translated to any specific architecture in a later phase.

Our particular SSE implementation shows highly competitive performance results, comparable to the ones achieved with the hand-coded low-level languages and extensions, such as OpenCL and SSE intrinsics, and substantially higher than those from pure compiler auto-vectorization technology. Even in those cases where compilers inhibit themselves from auto-vectorizing the code, our user-directed approach vectorizes the code and offers an important improvement in the execution time.

9.2 Future Work

We are actively working on this topic. Our future plans point towards adding support for new features like reduction operations, generic gather/scatter vector loads/stores, nested for-statements and if-statements, pattern recognition in expressions to be able to use compound vector instructions, like *dot product* and *madd*, among other functionalities.

With regards to supporting new SIMD extensions, AVX and LRBni are in our near future plans, although Neon and AltiVec extensions support are also in our medium-term work line. We are also thinking about extending this proposal to support accelerators in a wider way, using CUDA and OpenCL as output source code in our compiler.

Moving current Mercurium output code to a more compatible representation is also a priority task in order to be able to use different compilers as Mercurium back-end, like the Intel C/C++ Compiler.

We are currently working on a workshop and a conference papers as a result of this project. We are also developing and outlining a more formal and complete SIMD proposal that could be submitted to the OpenMP committee for its evaluation as part of the new version of the standard.

References

- [1] Aart J.C. Bik. The Software Vectorization Handbook: Applying Intel Multimedia Extensions for Maximum Performance. Intel Press, 2004.
- [2] Intel C++ Intrinsics Reference, Intel Corporation, 2007.
- [3] Intel SSE4 Programming Reference, Intel Corporation, 2007.
- [4] PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual. IBM Corporation, 2005.
- [5] Power ISA, Version 2.03. IBM Corporation, 2006.
- [6] Power ISA, Version 2.06, Revision B. IBM Corporation, 2010.
- [7] ARM Architecture Reference Manual. ARMv7-A and ARMv7-R Edition. ARM Corporation, 2011.
- [8] ARM Compiler Toolchain. Version 4.1. Assembler Reference. ARM Corporation, 2011.
- [9] Intel Advanced Vector Extensions Programming Reference. Intel Corporation, 2011.
- [10] AMD64 Architecture Programmer's Manual. Volume 4: 128-Bit and 256-Bit Media Instructions. Advanced Micro Devices Corporation, May 2011.
- [11] C++ Larrabee Prototype Library. Intel Corporation.
<http://software.intel.com/en-us/articles/prototype-primitives-guide/>
- [12] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Pradeep Dubey, Stephen Junkins, Adam Lake, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, Michael Abrash, Jeremy Sugerman, and Pat Hanrahan. Larrabee: A Manycore x86 Architecture for Visual Computing. In *IEEE Micro*, 29:10–21, January 2009.
- [13] Jonathan Parri, Daniel Shapiro, Miodrag Bolic and Voicu Groza. Returning Control to the Programmer: SIMD Intrinsics for Virtual Machines. In *ACM Queue*, 30:30-37, February 2011.
- [14] NVIDIA CUDA C Programming Guide, Version 4.0. NVIDIA Corporation, 2011.
- [15] The OpenCL Specification, Version 1.1. Khronos OpenCL Working Group, 2011.
- [16] Intel Array Building Blocks for Linux OS. User's Guide. Intel Corporation, 2011.
- [17] Kirk Skaugen. Petascale to Exascale. Extending Intel's HPC Commitment. Intel Corporation. In *International Supercomputing Conference*, 2010.
- [18] John Hennessy and David Patterson. Computer Architecture - A Quantitative Approach, 3th Edition. Morgan Kaufmann, 2003.

- [19] David W. Wall. Limits of Instruction-Level Parallelism. In *ASPLOS*, pages 176–188, 1991.
- [20] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *J. Supercomput.*, 7:9–50, May 1993.
- [21] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. In *IEEE Micro*, 17:12–19, September 1997.
- [22] OpenMP Application Program Interface, Version 3.1. July 2011.
- [23] Intel Threading Building Blocks Reference Manual. Intel Corporation, 2011.
- [24] Yaobin Wang, Hong An, Yuan Liu, Wanli Dong, and Kang Xu. Exploiting Speculative Thread-Level Parallelism Based on Transactional Memory. In *International Conference on Communications and Mobile Computing*, 137–140, 2011.
- [25] Eduard Ayguadé, Rosa M. Badía, Daniel Cabrera, Alejandro Duran, Marc González, Francisco D. Igual, Daniel Jiménez-González, Jesús Labarta, Xavier Martorell, Rafael Mayo, Josep M. Pérez, and Enrique S. Quintana-Ortí. A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. In *the International Workshops in OpenMP*, pages 154–167, 2009.
- [26] Judit Planas, Rosa M. Badía, Eduard Ayguadé, and Jesús Labarta. Hierarchical Task-Based Programming with StarSs. In *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.
- [27] Roger Espasa and Mateo Valero. Exploiting Instruction- and Data-Level Parallelism. In *IEEE Micro*, 17: 20–27, September 1997.
- [28] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 145–156, New York, NY, USA, 2000. ACM.
- [29] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The Illiac IV Computer. In *IEEE Transactional on Computation*, 17: 746–757, August 1968.
- [30] Altivec Technology Programming Environment Manual. Freescale Semiconductors, 2006.
- [31] Jeff Andrews and Nick Baker. Xbox 360 system architecture. In *IEEE Micro*, 26: 25–37, 2006.
- [32] Cray Inc. Corporate Webpage. <http://www.cray.com>
- [33] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. Evaluating Performance and Portability of OpenCL programs. In *The Fifth International Workshop on Automatic Performance Tuning*, June 2010.

-
- [34] Piotr Luszczek Stanimire Tomov Grefory Peterson Peng Du, Rick Weber and Jack Dongarra. From CUDA to OpenCL: Towards a Performance-Portable Solution for Multi-Platform. In *Parallel Computing*, (UT-CS-10-656):1–12, 2010.
 - [35] S. Rul, H. Vandierendonck, D’Haene J., and K. De Bosschere. An Experimental Study on Performance Portability of OpenCL Kernels. In *Symposium on Application Accelerators in High Performance Computing*, page 3, 2010.
 - [36] Intel C++ Compiler 12.0 User and Reference Guides.
http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011/compiler_c/index.htm
 - [37] The GCC Vector Extensions Website.
<http://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/Vector-Extensions.html#Vector-Extensions>
 - [38] Intel Math Kernel Library for Linux OS, Version 10.3. Intel Corporation, 2011.
 - [39] AMD Core Math Library (ACML), Version 4.4.0. Advanced Micro Devices Corporation, 2011.
 - [40] Peng Wu, Alexandre E. Eichenberger, Amy Wang, and Peng Zhao. An Integrated Simdization Framework Using Virtual Vectors. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS ’05, pages 169–178, New York, NY, USA, 2005. ACM.
 - [41] Alexandre E. Eichenberger, Peng Wu, and Kevin O’Brien. Vectorization for SIMD Architectures with Alignment Constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI’04, pages 82–93, New York, NY, USA, 2004. ACM.
 - [42] Peng Wu, Alexandre E. Eichenberger and Amy Wang. Efficient SIMD Code Generation for Runtime Alignment and Length Conversion. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO 2005.
 - [43] Ralf Karrenberg and Sebastian Hack. Whole-Function Vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO 2011.
 - [44] Manuel Hohenauer, Felix Engel, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A SIMD Optimization Framework for Retargetable Compilers. In *ACM Trans. Archit. Code Optim.*, 6:2:1–2:27, April 2009.
 - [45] Daniel McFarlin, Volodymyr Arbatov, Franz Franchetti, and Markus Püschel. Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets. In *International Conference on Supercomputing (ICS)*, 2011.
 - [46] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. Vapor SIMD: Auto-Vectorize Once, Run

- Everywhere. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO 2011 , pages 151–160. IEEE.
- [47] Dorit Nuzman and Richard Henderson. Multi-platform Auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.
 - [48] Dorit Naishlos, Marina Biberstein and Ayal Zaks. Compiler Vectorization Techniques for a Disjoint SIMD Architecture. In *PLDI* 2003.
 - [49] Dorit Nuzman, Ayal Zaks. Outer-Loop Vectorization – Revisited for Short SIMD Architectures. In *PACT* 2008.
 - [50] Dorit Naishlos, Marina Biberstein, Shay Ben-David and Ayal Zaks. Vectorizing for a SIMdD DSP Architecture. In *CASES* 2003.
 - [51] Jaewook Shin, Mary Hall and Jacqueline Chame. Superword-Level Parallelism in the Presence of Control Flow. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO 2005.
 - [52] Muhammad Omer Cheema and Omar Hammami. Application-specific SIMD Synthesis for Reconfigurable Architectures. *Microprocessors and Microsystems*, 30(6) :398–412, 2006.
 - [53] Alejandro Duran, Eduard Ayguadé, Rosa M. Badía, Jesús Labarta, Luis Martinell, Xavier Martorell and Judit Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. WSPC 2011.
 - [54] British Standards Institute. The C Standard. John Wiley and Sons Ltd, 2003.
 - [55] The Mercurium C/C++ Source-to-source Compiler Website.
<http://pm.bsc.es/projects/mcxx>
 - [56] The Nanos++ Runtime website. <http://pm.bsc.es/projects/nanox>
 - [57] The OmpSs Programming Model Website. <http://pm.bsc.es/projects/ompss>
 - [58] High-Level Transformations in Mercurium Compiler. User Manual, <http://pm.bsc.es/projects/mcxx/wiki/UserManual/HLT>. 2009.
 - [59] Intel OpenCL SDK 1.1. <http://software.intel.com/en-us/articles/opencl-sdk/>
 - [60] AMD APP OpenCL SDK 2.4
<http://developer.amd.com/sdks/AMDAPPSDK/Pages/default.aspx>
 - [61] NVIDIA CUDA SDK 4.0. <http://developer.nvidia.com/category/zone/cuda-zone>
 - [62] GCC 4.5.3 Compiler. <http://gcc.gnu.org/gcc-4.5/>
 - [63] IBM OpenCL SDK. <http://www.alphaworks.ibm.com/tech/opencl/>
 - [64] Intel Short Vector Math Library.
http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/cpp/win/intref_cls/common/intref_svml_overview.htm
 - [65] GNU Math Library. http://www.gnu.org/s/hello/manual/gnulib/math_002eh.html



Benchmarks Source Code

This appendix contains the kernel code of some of the benchmarks in the different states of the *simdization* process in the Mercurium compiler: annotated scalar code, intermediate generic code and GCC vector code for SSE. In order not to extremely increase the extension of the document, we only include the most complete benchmarks that illustrate the majority of the implemented features: Blacksholes and Perlin Noise.

A.1 Blacksholes

```
1  #pragma hlt simd
2  float phi(float X){
3      float y, absX, t;
4      const float c1 = 0.319381530f;
5      const float c2 = -0.356563782f;
6      const float c3 = 1.781477937f;
7      const float c4 = -1.821255978f;
8      const float c5 = 1.330274429f;
9      const float oneBySqrt2pi = 0.398942280f;
10
11     absX = fabsf(X);
12     t = 1.0f / (1.0f + 0.2316419f * absX);
13     y = 1.0f - oneBySqrt2pi * expf(-X * X / 2.0f) * t * (c1 + t * (c2 + t * (c3 + t * (c4 + t * c5))));
14     return (X < 0.0f) ? (1.0f - y) : y;
15 }
16
17 void blackScholesCPU(const int width, const int height, float* randArray, float* hostCallPrice,
18                    float* hostPutPrice){
19     int y;
20     #pragma hlt simd(randArray, hostCallPrice, hostPutPrice)
21     for (y = 0; y < width * height * 4; ++y) {
22         float d1, d2, sigmaSqrtT, float KexpMinusRT;
23         float s = S_LOWER_LIMIT * randArray[y] + S_UPPER_LIMIT * (1.0f - randArray[y]);
24         float k = K_LOWER_LIMIT * randArray[y] + K_UPPER_LIMIT * (1.0f - randArray[y]);
25         float t = T_LOWER_LIMIT * randArray[y] + T_UPPER_LIMIT * (1.0f - randArray[y]);
26         float r = R_LOWER_LIMIT * randArray[y] + R_UPPER_LIMIT * (1.0f - randArray[y]);
27         float sigma = SIGMA_LOWER_LIMIT * randArray[y] + SIGMA_UPPER_LIMIT * (1.0f - randArray[y]);
28
29         sigmaSqrtT = sigma * sqrtf(t);
30         d1 = (logf(s / k) + (r + sigma * sigma / 2.0f) * t) / sigmaSqrtT;
31         d2 = d1 - sigmaSqrtT;
32
33         KexpMinusRT = k * expf(-r * t);
34         hostCallPrice[y] = s * phi(d1) - KexpMinusRT * phi(d2);
35         hostPutPrice[y] = KexpMinusRT * phi(-d2) - s * phi(-d1);
36     }
37 }
```

Listing 16. Blacksholes kernel: annotated scalar code

```

1  void blackScholesCPU(const int width, const int height, float *randArray, float *hostCallPrice,
2                        float *hostPutPrice)
3  {
4      int y;
5      {
6          for (y = 0; y < width * height * 4; ++y)
7              {
8                  float __attribute__((generic_vector)) d1, d2;
9                  float __attribute__((generic_vector)) sigmaSqrtT;
10                 float __attribute__((generic_vector)) KexpMinusRT;
11                 float __attribute__((generic_vector)) s = __builtin_vector_expansion(10.0f) *
12                     __builtin_vector_reference(randArray[y]) + __builtin_vector_expansion(100.0f) *
13                     (__builtin_vector_expansion(1.0f) - __builtin_vector_reference(randArray[y]));
14                 float __attribute__((generic_vector)) k = __builtin_vector_expansion(10.0f) *
15                     __builtin_vector_reference(randArray[y]) + __builtin_vector_expansion(100.0f) *
16                     (__builtin_vector_expansion(1.0f) - __builtin_vector_reference(randArray[y]));
17                 float __attribute__((generic_vector)) t = __builtin_vector_expansion(1.0f) *
18                     __builtin_vector_reference(randArray[y]) + __builtin_vector_expansion(10.0f) *
19                     (__builtin_vector_expansion(1.0f) - __builtin_vector_reference(randArray[y]));
20                 float __attribute__((generic_vector)) r = __builtin_vector_expansion(0.01f) *
21                     __builtin_vector_reference(randArray[y]) + __builtin_vector_expansion(0.05f) *
22                     (__builtin_vector_expansion(1.0f) - __builtin_vector_reference(randArray[y]));
23                 float __attribute__((generic_vector)) sigma = __builtin_vector_expansion(0.01f) *
24                     __builtin_vector_reference(randArray[y]) + __builtin_vector_expansion(0.10f) *
25                     (__builtin_vector_expansion(1.0f) - __builtin_vector_reference(randArray[y]));
26
27                 sigmaSqrtT = sigma * __builtin_generic_function(sqrtf, t);
28                 d1 = (__builtin_generic_function(logf, s / k) + (r + sigma * sigma /
29                     __builtin_vector_expansion(2.0f)) * t) / sigmaSqrtT;
30                 d2 = d1 - sigmaSqrtT;
31
32                 KexpMinusRT = k * __builtin_generic_function(expf, -r * t);
33                 __builtin_vector_reference(hostCallPrice[y]) = s * __builtin_generic_function(phi, d1) -
34                     KexpMinusRT * __builtin_generic_function(phi, d2);
35                 __builtin_vector_reference(hostPutPrice[y]) = KexpMinusRT * __builtin_generic_function(phi, -d2)
36                     - s * __builtin_generic_function(phi, -d1);
37             }
38         for (; y < width * height * 4; ++y)
39             {
40                 float d1, d2;
41                 float sigmaSqrtT;
42                 float KexpMinusRT;
43                 float s = 10.0f * randArray[y] + 100.0f * (1.0f - randArray[y]);
44                 float k = 10.0f * randArray[y] + 100.0f * (1.0f - randArray[y]);
45                 float t = 1.0f * randArray[y] + 10.0f * (1.0f - randArray[y]);
46                 float r = 0.01f * randArray[y] + 0.05f * (1.0f - randArray[y]);
47                 float sigma = 0.01f * randArray[y] + 0.10f * (1.0f - randArray[y]);
48
49                 sigmaSqrtT = sigma * sqrtf(t);
50                 d1 = (logf(s / k) + (r + sigma * sigma / 2.0f) * t) / sigmaSqrtT;
51                 d2 = d1 - sigmaSqrtT;
52
53                 KexpMinusRT = k * expf(-r * t);
54                 hostCallPrice[y] = s * phi(d1) - KexpMinusRT * phi(d2);
55                 hostPutPrice[y] = KexpMinusRT * phi(-d2) - s * phi(-d1);
56             }
57     }
58 }

```

Listing 17. Blacksholes kernel: intermediate generic code

```

1  float phi(float X)
2  {
3      float y, absX, t;
4      const float c1 = 0.319381530f;
5      const float c2 = -0.356563782f;
6      const float c3 = 1.781477937f;
7      const float c4 = -1.821255978f;
8      const float c5 = 1.330274429f;
9      const float oneBySqrt2pi = 0.398942280f;
10
11     absX = fabsf(X);
12     t = 1.0f / (1.0f + 0.2316419f * absX);
13     y = 1.0f - oneBySqrt2pi * expf(-X * X / 2.0f) * t * (c1 + t * (c2 + t * (c3 + t * (c4 + t * c5))));
14     return (X < 0.0f) ? (1.0f - y) : y;
15 }
16
17 float __attribute__((vector_size(16))) _phi_smp_16(float __attribute__((vector_size(16))) X)
18 {
19     float __attribute__((vector_size(16))) y, absX, t;
20     const float __attribute__((vector_size(16))) c1 =
21         ((float __attribute__((vector_size(16)))){0.319381530f, 0.319381530f, 0.319381530f, 0.319381530f});
22     const float __attribute__((vector_size(16))) c2 =
23         ((float __attribute__((vector_size(16)))){-0.356563782f, -0.356563782f, -0.356563782f, -0.356563782f});
24     const float __attribute__((vector_size(16))) c3 =
25         ((float __attribute__((vector_size(16)))){1.781477937f, 1.781477937f, 1.781477937f, 1.781477937f});
26     const float __attribute__((vector_size(16))) c4 =
27         ((float __attribute__((vector_size(16)))){-1.821255978f, -1.821255978f, -1.821255978f, -1.821255978f});
28     const float __attribute__((vector_size(16))) c5 =
29         ((float __attribute__((vector_size(16)))){1.330274429f, 1.330274429f, 1.330274429f, 1.330274429f});
30     const float __attribute__((vector_size(16))) oneBySqrt2pi =
31         ((float __attribute__((vector_size(16)))){0.398942280f, 0.398942280f, 0.398942280f, 0.398942280f});
32
33     absX = __fabsf_default_smp_16(X);
34     t = ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}) /
35         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}) +
36         ((float __attribute__((vector_size(16)))){0.2316419f, 0.2316419f, 0.2316419f, 0.2316419f}) * absX;
37     y = ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}) - oneBySqrt2pi * __vrs4_exp(-X *
38         X / ((float __attribute__((vector_size(16)))){2.0f, 2.0f, 2.0f, 2.0f})) * t * (c1 + t * (c2 + t *
39         (c3 + t * (c4 + t * c5))));
40     return __builtin_ia32_blendvps(y, ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}) - y,
41         ((float __attribute__((vector_size(16)))) __builtin_ia32_cmltps(X,
42         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})));
43 }
44
45 static inline float __attribute__((vector_size(16))) __fabsf_default_smp_16(
46     float __attribute__((vector_size(16))) a)
47 {
48     return (float __attribute__((vector_size(16)))) ((int __attribute__((vector_size(16)))) a) &
49         ((int __attribute__((vector_size(16)))){0x7FFFFFFF, 0x7FFFFFFF, 0x7FFFFFFF, 0x7FFFFFFF});
50 }
51
52 void __attribute__((noinline)) blackScholesCPU(const int width, const int height, float *randArray,
53     float *hostCallPrice, float *hostPutPrice)
54 {
55     int y;
56     {
57         for (y = 0; y < ((width * height * 4) - (4 - 1)); y += 4)
58         {
59             float __attribute__((vector_size(16))) d1, d2, sigmaSqrtT, KexpMinusRT;
60             float __attribute__((vector_size(16))) s =
61                 ((float __attribute__((vector_size(16)))){10.0f, 10.0f, 10.0f, 10.0f}) *
62                 *((float __attribute__((vector_size(16)))) (*) &(randArray[y])) +
63                 ((float __attribute__((vector_size(16)))){100.0f, 100.0f, 100.0f, 100.0f}) *
64                 ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}) - *
65                 ((float __attribute__((vector_size(16)))) (*) &(randArray[y]));
66             float __attribute__((vector_size(16))) k =
67                 ((float __attribute__((vector_size(16)))){10.0f, 10.0f, 10.0f, 10.0f}) *
68                 *((float __attribute__((vector_size(16)))) (*) &(randArray[y])) +
69                 ((float __attribute__((vector_size(16)))){100.0f, 100.0f, 100.0f, 100.0f}) *
70                 ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}) -
71                 *((float __attribute__((vector_size(16)))) (*) &(randArray[y]));
72             float __attribute__((vector_size(16))) t =
73                 ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}) *
74                 *((float __attribute__((vector_size(16)))) (*) &(randArray[y])) +
75                 ((float __attribute__((vector_size(16)))){10.0f, 10.0f, 10.0f, 10.0f}) *

```

```

76         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}) -
77         *((float __attribute__((vector_size(16)))) (*) &(randArray[y]));
78     float __attribute__((vector_size(16))) r =
79     ((float __attribute__((vector_size(16)))){0.01f, 0.01f, 0.01f, 0.01f}) *
80     *((float __attribute__((vector_size(16)))) (*) &(randArray[y])) +
81     ((float __attribute__((vector_size(16)))){0.05f, 0.05f, 0.05f, 0.05f}) *
82     ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}) -
83     *((float __attribute__((vector_size(16)))) (*) &(randArray[y]));
84     float __attribute__((vector_size(16))) sigma =
85     ((float __attribute__((vector_size(16)))){0.01f, 0.01f, 0.01f, 0.01f}) *
86     *((float __attribute__((vector_size(16)))) (*) &(randArray[y])) +
87     ((float __attribute__((vector_size(16)))){0.10f, 0.10f, 0.10f, 0.10f}) *
88     ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}) -
89     *((float __attribute__((vector_size(16)))) (*) &(randArray[y]));
90
91     sigmaSqrtT = sigma * __builtin_ia32_sqrtfs(t);
92     d1 = (__vrs4_logf(s / k) + (r + sigma * sigma /
93     ((float __attribute__((vector_size(16)))){2.0f, 2.0f, 2.0f, 2.0f})) * t) / sigmaSqrtT;
94     d2 = d1 - sigmaSqrtT;
95
96     KexpMinusRT = k * __vrs4_expfs(-r * t);
97     *((float __attribute__((vector_size(16)))) (*) &(hostCallPrice[y])) = s * _phi_smp_16(d1) -
98     KexpMinusRT * _phi_smp_16(d2);
99     *((float __attribute__((vector_size(16)))) (*) &(hostPutPrice[y])) = KexpMinusRT *
100     _phi_smp_16(-d2) - s * _phi_smp_16(-d1);
101 }
102 for (; y < width * height * 4; ++y)
103 {
104     float d1, d2;
105     float sigmaSqrtT;
106     float KexpMinusRT;
107     float s = 10.0f * randArray[y] + 100.0f * (1.0f - randArray[y]);
108     float k = 10.0f * randArray[y] + 100.0f * (1.0f - randArray[y]);
109     float t = 1.0f * randArray[y] + 10.0f * (1.0f - randArray[y]);
110     float r = 0.01f * randArray[y] + 0.05f * (1.0f - randArray[y]);
111     float sigma = 0.01f * randArray[y] + 0.10f * (1.0f - randArray[y]);
112
113     sigmaSqrtT = sigma * sqrtf(t);
114     d1 = (logf(s / k) + (r + sigma * sigma / 2.0f) * t) / sigmaSqrtT;
115     d2 = d1 - sigmaSqrtT;
116
117     KexpMinusRT = k * expf(-r * t);
118     hostCallPrice[y] = s * phi(d1) - KexpMinusRT * phi(d2);
119     hostPutPrice[y] = KexpMinusRT * phi(-d2) - s * phi(-d1);
120 }
121 }
122 }

```

Listing 18. Blacksholes kernel: GCC vector extensions with ACML

A.2 Perlin Noise

```

1  #pragma hlt simd
2  float lerp(float t, float a, float b)
3  {
4      return a + t * (b - a);
5  }
6
7  #pragma hlt simd
8  float grad(int hash, float x, float y, float z)
9  {
10     int h = hash & 15;          /* Convert low 4 bits of hash code */
11     float u = (h < 8) ? x : y;   /* into 12 gradient directions. */
12     float v = (h < 4) ? y : ((h == 12) || (h == 14)) ? x : z;
13
14     u = (h & 1) == 0 ? u : -u;
15     v = (h & 2) == 0 ? v : -v;
16     return u + v;
17 }

```

```

18  #pragma hlt simd
19  float noise3(float x, float y, float z)
20  {
21      float floor_x = floorf(x);
22      float floor_y = floorf(y);
23      float floor_z = floorf(z);
24
25      int X = (int) floor_x & 255; /* Find unit cube that */
26      int Y = (int) floor_y & 255; /* contains point. */
27      int Z = (int) floor_z & 255;
28
29      x -= floor_x; /* Find relative x,y,z */
30      y -= floor_y; /* of point in cube. */
31      z -= floor_z;
32
33      float x1 = x - 1.0f;
34      float y1 = y - 1.0f;
35      float z1 = z - 1.0f;
36
37      float u = fade(x); /* Compute fade curves */
38      float v = fade(y); /* for each of x,y,z. */
39      float w = fade(z);
40
41      int A = perm[X] + Y;
42      int AA = perm[A] + Z;
43      int AB = perm[A + 1] + Z; /* Hash coordinates of */
44      int B = perm[X + 1] + Y; /* the 8 cube corners. */
45      int BA = perm[B] + Z;
46      int BB = perm[B + 1] + Z;
47
48      float g0 = grad(perm[AA], x, y, z);
49      float g1 = grad(perm[BA], x1, y, z);
50      float g2 = grad(perm[AB], x, y1, z);
51      float g3 = grad(perm[BB], x1, y1, z);
52      float g4 = grad(perm[AA + 1], x, y, z1);
53      float g5 = grad(perm[BA + 1], x1, y, z1);
54      float g6 = grad(perm[AB + 1], x, y1, z1);
55      float g7 = grad(perm[BB + 1], x1, y1, z1);
56
57      /* Add blended results from 8 corners of cube. */
58      float u01 = lerp(u, g0, g1);
59      float u23 = lerp(u, g2, g3);
60      float u45 = lerp(u, g4, g5);
61      float u67 = lerp(u, g6, g7);
62
63      float v0 = lerp(v, u01, u23);
64      float v1 = lerp(v, u45, u67);
65
66      return lerp(w, v0, v1);
67  }
68
69  void compute_perlin_noise(unsigned char * output_red, unsigned char * output_green,
70                          unsigned char * output_blue, unsigned char * output_alpha, const float time,
71                          const unsigned int rowstride, const int img_height, const int img_width)
72  {
73      unsigned int i, j;
74      const float vdx = 0.03125f;
75      const float vdy = 0.0125f;
76      const float vs = 2.0f;
77      const float bias = 0.35f;
78
79      for (j = 0; j < img_height; j++) {
80          #pragma hlt simd (output_red, output_green, output_blue, output_alpha)
81          for (i = 0; i < img_width; i++) {
82
83              float vx, vy, vt;
84              float red, green, blue;
85              float xx, yy;
86
87              vx = i * vdx;
88              vy = j * vdy;
89              vt = time * vs;
90
91              xx = vx * vs;
92              yy = vy * vs;
93
94              red = noise3(xx, vt, yy);
95              green = noise3(vt, yy, xx);
96              blue = noise3(yy, xx, vt);

```

```
97
98         red += bias;
99         green += bias;
100        blue += bias;
101
102        // Clamp to within [0 .. 1]
103        red = (red > 1.0f) ? 1.0f : red;
104        green = (green > 1.0f) ? 1.0f : green;
105        blue = (blue > 1.0f) ? 1.0f : blue;
106
107        red = (red < 0.0f) ? 0.0f : red;
108        green = (green < 0.0f) ? 0.0f : green;
109        blue = (blue < 0.0f) ? 0.0f : blue;
110
111        red *= 255.0f;
112        green *= 255.0f;
113        blue *= 255.0f;
114
115        output_red[(j * rowstride) + i] = red;
116        output_green[(j * rowstride) + i] = green;
117        output_blue[(j * rowstride) + i] = blue;
118        output_alpha[(j * rowstride) + i] = 255;
119    }
120 }
121 }
```

Listing 19. Perlin Noise kernel: annotated scalar code

```
1  void compute_perlin_noise(unsigned char *output_red, unsigned char *output_green,
2                          unsigned char *output_blue, unsigned char *output_alpha, const float time,
3                          const unsigned int rowstride, const int img_height, const int img_width)
4  {
5      unsigned int i, j;
6      const float vdx = 0.03125f;
7      const float vdy = 0.0125f;
8      const float vs = 2.0f;
9      const float bias = 0.35f;
10
11      for (j = 0; j < img_height; j++)
12      {
13          {
14              for (i = 0; i < img_width; i++)
15              {
16                  float __attribute__((generic_vector)) vx, vy, vt;
17                  float __attribute__((generic_vector)) red, green, blue;
18                  float __attribute__((generic_vector)) xx, yy;
19                  vx = __builtin_vector_conversion(__builtin_ind_var_vector_expansion(i),
20                                                  __builtin_vector_expansion(vdx)) * __builtin_vector_expansion(vdx);
21                  vy = __builtin_vector_conversion(__builtin_vector_expansion(j),
22                                                  __builtin_vector_expansion(vdy)) * __builtin_vector_expansion(vdy);
23                  vt = __builtin_vector_expansion(time) * __builtin_vector_expansion(vs);
24                  xx = vx * __builtin_vector_expansion(vs);
25                  yy = vy * __builtin_vector_expansion(vs);
26                  red = __builtin_generic_function(noise3, xx, vt, yy);
27                  green = __builtin_generic_function(noise3, vt, yy, xx);
28                  blue = __builtin_generic_function(noise3, yy, xx, vt);
29                  red += __builtin_vector_expansion(bias);
30                  green += __builtin_vector_expansion(bias);
31                  blue += __builtin_vector_expansion(bias);
32                  red = (red > __builtin_vector_expansion(1.0f)) ? __builtin_vector_expansion(1.0f) : red;
33                  green = (green > __builtin_vector_expansion(1.0f)) ? __builtin_vector_expansion(1.0f) : green;
34                  blue = (blue > __builtin_vector_expansion(1.0f)) ? __builtin_vector_expansion(1.0f) : blue;
35                  red = (red < __builtin_vector_expansion(0.0f)) ? __builtin_vector_expansion(0.0f) : red;
36                  green = (green < __builtin_vector_expansion(0.0f)) ? __builtin_vector_expansion(0.0f) : green;
37                  blue = (blue < __builtin_vector_expansion(0.0f)) ? __builtin_vector_expansion(0.0f) : blue;
38                  red *= __builtin_vector_expansion(255.0f);
39                  green *= __builtin_vector_expansion(255.0f);
40                  blue *= __builtin_vector_expansion(255.0f);
41                  __builtin_vector_reference(output_red[(j * rowstride) + i]) =
42                      __builtin_vector_conversion(red, __builtin_vector_reference(output_red[j*rowstride+i]));
43                  __builtin_vector_reference(output_green[(j * rowstride) + i]) =
44                      __builtin_vector_conversion(green,
```

```

45     __builtin_vector_reference(output_green[j*rowstride+i]));
46     __builtin_vector_reference(output_blue[(j * rowstride) + i]) =
47     __builtin_vector_conversion(blue,
48     __builtin_vector_reference(output_blue[j*rowstride+i]));
49     __builtin_vector_reference(output_alpha[j*rowstride+i]) =
50     __builtin_vector_conversion(__builtin_vector_expansion(255),
51     __builtin_vector_reference(output_alpha[j*rowstride+i]));
52 }
53 for (; i < img_width; i++)
54 {
55     float vx, vy, vt;
56     float red, green, blue;
57     float xx, yy;
58     vx = i * vdx;
59     vy = j * vdy;
60     vt = time * vs;
61     xx = vx * vs;
62     yy = vy * vs;
63     red = noise3(xx, vt, yy);
64     green = noise3(vt, yy, xx);
65     blue = noise3(yy, xx, vt);
66     red += bias;
67     green += bias;
68     blue += bias;
69     red = (red > 1.0f) ? 1.0f : red;
70     green = (green > 1.0f) ? 1.0f : green;
71     blue = (blue > 1.0f) ? 1.0f : blue;
72     red = (red < 0.0f) ? 0.0f : red;
73     green = (green < 0.0f) ? 0.0f : green;
74     blue = (blue < 0.0f) ? 0.0f : blue;
75     red *= 255.0f;
76     green *= 255.0f;
77     blue *= 255.0f;
78     output_red[(j * rowstride) + i] = red;
79     output_green[(j * rowstride) + i] = green;
80     output_blue[(j * rowstride) + i] = blue;
81     output_alpha[(j * rowstride) + i] = 255;
82 }
83 }
84 }
85 }

```

Listing 20. Perlin Noise kernel: intermediate generic code

```

1  float __attribute__((vector_size(16))) _noise3_smp_16(float __attribute__((vector_size(16))) x,
2  float __attribute__((vector_size(16))) y, float __attribute__((vector_size(16))) z)
3  {
4      float __attribute__((vector_size(16))) floor_x = __floorf_default_smp_16(x);
5      float __attribute__((vector_size(16))) floor_y = __floorf_default_smp_16(y);
6      float __attribute__((vector_size(16))) floor_z = __floorf_default_smp_16(z);
7      int __attribute__((vector_size(16))) X = __conv_float_to_int_smp16(floor_x) &
8      ((int __attribute__((vector_size(16))))(255, 255, 255, 255));
9      int __attribute__((vector_size(16))) Y = __conv_float_to_int_smp16(floor_y) &
10     ((int __attribute__((vector_size(16))))(255, 255, 255, 255));
11     int __attribute__((vector_size(16))) Z = __conv_float_to_int_smp16(floor_z) &
12     ((int __attribute__((vector_size(16))))(255, 255, 255, 255));
13     x -= floor_x;
14     y -= floor_y;
15     z -= floor_z;
16     float __attribute__((vector_size(16))) x1 = x -
17     ((float __attribute__((vector_size(16))))(1.0f, 1.0f, 1.0f, 1.0f));
18     float __attribute__((vector_size(16))) y1 = y -
19     ((float __attribute__((vector_size(16))))(1.0f, 1.0f, 1.0f, 1.0f));
20     float __attribute__((vector_size(16))) z1 = z -
21     ((float __attribute__((vector_size(16))))(1.0f, 1.0f, 1.0f, 1.0f));
22     float __attribute__((vector_size(16))) u = _fade_smp_16(x);
23     float __attribute__((vector_size(16))) v = _fade_smp_16(y);
24     float __attribute__((vector_size(16))) w = _fade_smp_16(z);
25     int __attribute__((vector_size(16))) A = __vector_subscript(perm, X) + Y;
26     int __attribute__((vector_size(16))) AA = __vector_subscript(perm, A) + Z;
27     int __attribute__((vector_size(16))) AB = __vector_subscript(perm, A +
28     ((int __attribute__((vector_size(16))))(1, 1, 1, 1))) + Z;

```

Appendix A: Benchmarks Source Code

```
29     int __attribute__((vector_size(16))) B = __vector_subscript(perm, X +
30         ((int __attribute__((vector_size(16)))){1, 1, 1, 1})) + Y;
31     int __attribute__((vector_size(16))) BA = __vector_subscript(perm, B) + Z;
32     int __attribute__((vector_size(16))) BB = __vector_subscript(perm, B +
33         ((int __attribute__((vector_size(16)))){1, 1, 1, 1})) + Z;
34     float __attribute__((vector_size(16))) g0 = _grad_smp_16(__vector_subscript(perm, AA), x, y, z);
35     float __attribute__((vector_size(16))) g1 = _grad_smp_16(__vector_subscript(perm, BA), x1, y, z);
36     float __attribute__((vector_size(16))) g2 = _grad_smp_16(__vector_subscript(perm, AB), x, y1, z);
37     float __attribute__((vector_size(16))) g3 = _grad_smp_16(__vector_subscript(perm, BB), x1, y1, z);
38     float __attribute__((vector_size(16))) g4 = _grad_smp_16(__vector_subscript(perm, AA +
39         ((int __attribute__((vector_size(16)))){1, 1, 1, 1})), x, y, z1);
40     float __attribute__((vector_size(16))) g5 = _grad_smp_16(__vector_subscript(perm, BA +
41         ((int __attribute__((vector_size(16)))){1, 1, 1, 1})), x1, y, z1);
42     float __attribute__((vector_size(16))) g6 = _grad_smp_16(__vector_subscript(perm, AB +
43         ((int __attribute__((vector_size(16)))){1, 1, 1, 1})), x, y1, z1);
44     float __attribute__((vector_size(16))) g7 = _grad_smp_16(__vector_subscript(perm, BB +
45         ((int __attribute__((vector_size(16)))){1, 1, 1, 1})), x1, y1, z1);
46     float __attribute__((vector_size(16))) u01 = _lerp_smp_16(u, g0, g1);
47     float __attribute__((vector_size(16))) u23 = _lerp_smp_16(u, g2, g3);
48     float __attribute__((vector_size(16))) u45 = _lerp_smp_16(u, g4, g5);
49     float __attribute__((vector_size(16))) u67 = _lerp_smp_16(u, g6, g7);
50     float __attribute__((vector_size(16))) v0 = _lerp_smp_16(v, u01, u23);
51     float __attribute__((vector_size(16))) v1 = _lerp_smp_16(v, u45, u67);
52     return _lerp_smp_16(w, v0, v1);
53 }
54
55 static inline unsigned char __attribute__((vector_size(16))) __conv_float_to_uchar_smp16(
56     float __attribute__((vector_size(16))) vf0, float __attribute__((vector_size(16))) vf1,
57     float __attribute__((vector_size(16))) vf2, float __attribute__((vector_size(16))) vf3)
58 {
59     int __attribute__((vector_size(16))) vi0, vi1;
60     short int __attribute__((vector_size(16))) vs0, vs1;
61     vi0 = __builtin_ia32_cvttps2dq(vf0);
62     vi1 = __builtin_ia32_cvttps2dq(vf1);
63     vs0 = __builtin_ia32_packusdw128(vi0, vi1);
64     vi0 = __builtin_ia32_cvttps2dq(vf2);
65     vi1 = __builtin_ia32_cvttps2dq(vf3);
66     vs1 = __builtin_ia32_packusdw128(vi0, vi1);
67     return (unsigned char __attribute__((vector_size(16)))) __builtin_ia32_packuswb128(vs0, vs1);
68 }
69
70 static inline int __attribute__((vector_size(16))) __conv_float_to_int_smp16(
71     float __attribute__((vector_size(16))) vf)
72 {
73     return __builtin_ia32_cvttps2dq(vf);
74 }
75
76 static inline unsigned char __attribute__((vector_size(16))) __conv_int_to_uchar_smp16(
77     int __attribute__((vector_size(16))) vi0, int __attribute__((vector_size(16))) vi1,
78     int __attribute__((vector_size(16))) vi2, int __attribute__((vector_size(16))) vi3)
79 {
80     short int __attribute__((vector_size(16))) vs0, vs1;
81     vs0 = __builtin_ia32_packusdw128(vi0, vi1);
82     vs1 = __builtin_ia32_packusdw128(vi2, vi3);
83     return (unsigned char __attribute__((vector_size(16)))) __builtin_ia32_packuswb128(vs0, vs1);
84 }
85
86 static inline float __attribute__((vector_size(16))) __conv_uint_to_float_smp16(
87     unsigned int __attribute__((vector_size(16))) vi)
88 {
89     return __builtin_ia32_cvtqd2ps((int __attribute__((vector_size(16)))) vi);
90 }
91
92 static inline int __attribute__((vector_size(16))) __vector_subscript(
93     int subscripted[], int __attribute__((vector_size(16))) subscript)
94 {
95     int __attribute__((vector_size(16))) result =
96         (int __attribute__((vector_size(16)))){subscripted[__builtin_ia32_vec_ext_v4si(subscript, 0)],
97         subscripted[__builtin_ia32_vec_ext_v4si(subscript, 1)],
98         subscripted[__builtin_ia32_vec_ext_v4si(subscript, 2)],
99         subscripted[__builtin_ia32_vec_ext_v4si(subscript, 3)]};
100     return result;
101 }
102
103 static inline float __attribute__((vector_size(16))) __floorf_default_smp_16(
104     float __attribute__((vector_size(16))) a)
105 {
106     return __builtin_ia32_roundps(a, 0x01 | 0x00);
107 }
```

```

108
109 float __attribute__((vector_size(16))) _fade_smp16(
110     float __attribute__((vector_size(16))) t)
111 {
112     return t * t * t * (t * (t * ((float __attribute__((vector_size(16)))){6.0f, 6.0f, 6.0f, 6.0f}) -
113         ((float __attribute__((vector_size(16)))){15.0f, 15.0f, 15.0f, 15.0f})) +
114         ((float __attribute__((vector_size(16)))){10.0f, 10.0f, 10.0f, 10.0f}));
115 }
116
117 float __attribute__((vector_size(16))) _lerp_smp16(float __attribute__((vector_size(16))) t,
118     float __attribute__((vector_size(16))) a, float __attribute__((vector_size(16))) b)
119 {
120     return a + t * (b - a);
121 }
122
123 float __attribute__((vector_size(16))) _grad_smp16(int __attribute__((vector_size(16))) hash,
124     float __attribute__((vector_size(16))) x, float __attribute__((vector_size(16))) y,
125     float __attribute__((vector_size(16))) z)
126 {
127     int __attribute__((vector_size(16))) h = hash &
128         ((int __attribute__((vector_size(16)))){15, 15, 15, 15});
129     float __attribute__((vector_size(16))) u = _builtin_ia32_blendvps(y, x,
130         ((float __attribute__((vector_size(16)))) (_builtin_ia32_pcmptgt128(
131             ((int __attribute__((vector_size(16)))){8, 8, 8, 8}), h)));
132     float __attribute__((vector_size(16))) v = _builtin_ia32_blendvps(
133         _builtin_ia32_blendvps(z, x, ((float __attribute__((vector_size(16))))
134             (_builtin_ia32_pcmptgt128(h, ((int __attribute__((vector_size(16)))){12, 12, 12, 12})) |
135             _builtin_ia32_pcmptgt128(h, ((int __attribute__((vector_size(16)))){14, 14, 14, 14})))), y,
136         ((float __attribute__((vector_size(16)))) (_builtin_ia32_pcmptgt128(
137             ((int __attribute__((vector_size(16)))){4, 4, 4, 4}), h)));
138     u = _builtin_ia32_blendvps(-u, u, ((float __attribute__((vector_size(16))))
139         (_builtin_ia32_pcmptgt128(h & ((int __attribute__((vector_size(16)))){1, 1, 1, 1}),
140             ((int __attribute__((vector_size(16)))){0, 0, 0, 0})))));
141     v = _builtin_ia32_blendvps(-v, v, ((float __attribute__((vector_size(16))))
142         (_builtin_ia32_pcmptgt128(h & ((int __attribute__((vector_size(16)))){2, 2, 2, 2}),
143             ((int __attribute__((vector_size(16)))){0, 0, 0, 0})))));
144     return u + v;
145 }
146
147 void compute_perlin_noise(unsigned char *output_red, unsigned char *output_green,
148     unsigned char *output_blue, unsigned char *output_alpha, const float time,
149     const unsigned int rowstride, const int img_height, const int img_width)
150 {
151     unsigned int i, j;
152     const float vdx = 0.03125f;
153     const float vdy = 0.0125f;
154     const float vs = 2.0f;
155     const float bias = 0.35f;
156     for (j = 0; j < img_height; j++)
157     {
158         {
159             for (i = 0; i < ((img_width) - (16 - 1)); i += 16)
160             {
161                 float __attribute__((vector_size(16))) _vx_rep0, _vy_rep0, _vt_rep0;
162                 float __attribute__((vector_size(16))) _vx_rep1, _vy_rep1, _vt_rep1;
163                 float __attribute__((vector_size(16))) _vx_rep2, _vy_rep2, _vt_rep2;
164                 float __attribute__((vector_size(16))) _vx_rep3, _vy_rep3, _vt_rep3;
165                 float __attribute__((vector_size(16))) _red_rep0, _green_rep0, _blue_rep0;
166                 float __attribute__((vector_size(16))) _red_rep1, _green_rep1, _blue_rep1;
167                 float __attribute__((vector_size(16))) _red_rep2, _green_rep2, _blue_rep2;
168                 float __attribute__((vector_size(16))) _red_rep3, _green_rep3, _blue_rep3;
169                 float __attribute__((vector_size(16))) _xx_rep0, _yy_rep0;
170                 float __attribute__((vector_size(16))) _xx_rep1, _yy_rep1;
171                 float __attribute__((vector_size(16))) _xx_rep2, _yy_rep2;
172                 float __attribute__((vector_size(16))) _xx_rep3, _yy_rep3;
173                 _vx_rep0 = __conv_uint_to_float_smp16(
174                     ((unsigned int __attribute__((vector_size(16)))){i, i, i, i} +
175                     (unsigned int __attribute__((vector_size(16)))){0, 1, 2, 3})) *
176                     ((const float __attribute__((vector_size(16)))){vdx, vdx, vdx, vdx});
177                 _vx_rep1 = __conv_uint_to_float_smp16(
178                     ((unsigned int __attribute__((vector_size(16)))){i, i, i, i} +
179                     (unsigned int __attribute__((vector_size(16)))){4, 5, 6, 7})) *
180                     ((const float __attribute__((vector_size(16)))){vdx, vdx, vdx, vdx});
181                 _vx_rep2 = __conv_uint_to_float_smp16(
182                     ((unsigned int __attribute__((vector_size(16)))){i, i, i, i} +
183                     (unsigned int __attribute__((vector_size(16)))){8, 9, 10, 11})) *
184                     ((const float __attribute__((vector_size(16)))){vdx, vdx, vdx, vdx});
185                 _vx_rep3 = __conv_uint_to_float_smp16(
186                     ((unsigned int __attribute__((vector_size(16)))){i, i, i, i} +

```

Appendix A: Benchmarks Source Code

```
187         (unsigned int __attribute__((vector_size(16)))){12, 13, 14, 15})) *
188         ((const float __attribute__((vector_size(16)))){vdx, vdx, vdx, vdx}));
189     _vy_rep0 = __conv_uint_to_float_smp16(
190         ((unsigned int __attribute__((vector_size(16)))){j, j, j, j})) *
191         ((const float __attribute__((vector_size(16)))){vdy, vdy, vdy, vdy}));
192     _vy_rep1 = __conv_uint_to_float_smp16(
193         ((unsigned int __attribute__((vector_size(16)))){j, j, j, j})) *
194         ((const float __attribute__((vector_size(16)))){vdy, vdy, vdy, vdy}));
195     _vy_rep2 = __conv_uint_to_float_smp16(
196         ((unsigned int __attribute__((vector_size(16)))){j, j, j, j})) *
197         ((const float __attribute__((vector_size(16)))){vdy, vdy, vdy, vdy}));
198     _vy_rep3 = __conv_uint_to_float_smp16(
199         ((unsigned int __attribute__((vector_size(16)))){j, j, j, j})) *
200         ((const float __attribute__((vector_size(16)))){vdy, vdy, vdy, vdy}));
201     _vt_rep0 = ((const float __attribute__((vector_size(16)))){time, time, time, time}) *
202         ((const float __attribute__((vector_size(16)))){vs, vs, vs, vs});
203     _vt_rep1 = ((const float __attribute__((vector_size(16)))){time, time, time, time}) *
204         ((const float __attribute__((vector_size(16)))){vs, vs, vs, vs});
205     _vt_rep2 = ((const float __attribute__((vector_size(16)))){time, time, time, time}) *
206         ((const float __attribute__((vector_size(16)))){vs, vs, vs, vs});
207     _vt_rep3 = ((const float __attribute__((vector_size(16)))){time, time, time, time}) *
208         ((const float __attribute__((vector_size(16)))){vs, vs, vs, vs});
209     _xx_rep0 = _vx_rep0 * ((const float __attribute__((vector_size(16)))){vs, vs, vs, vs});
210     _xx_rep1 = _vx_rep1 * ((const float __attribute__((vector_size(16)))){vs, vs, vs, vs});
211     _xx_rep2 = _vx_rep2 * ((const float __attribute__((vector_size(16)))){vs, vs, vs, vs});
212     _xx_rep3 = _vx_rep3 * ((const float __attribute__((vector_size(16)))){vs, vs, vs, vs});
213     _yy_rep0 = _vy_rep0 * ((const float __attribute__((vector_size(16)))){vs, vs, vs, vs});
214     _yy_rep1 = _vy_rep1 * ((const float __attribute__((vector_size(16)))){vs, vs, vs, vs});
215     _yy_rep2 = _vy_rep2 * ((const float __attribute__((vector_size(16)))){vs, vs, vs, vs});
216     _yy_rep3 = _vy_rep3 * ((const float __attribute__((vector_size(16)))){vs, vs, vs, vs});
217     _red_rep0 = _noise3_smp_16(_xx_rep0, _vt_rep0, _yy_rep0);
218     _red_rep1 = _noise3_smp_16(_xx_rep1, _vt_rep1, _yy_rep1);
219     _red_rep2 = _noise3_smp_16(_xx_rep2, _vt_rep2, _yy_rep2);
220     _red_rep3 = _noise3_smp_16(_xx_rep3, _vt_rep3, _yy_rep3);
221     _green_rep0 = _noise3_smp_16(_vt_rep0, _yy_rep0, _xx_rep0);
222     _green_rep1 = _noise3_smp_16(_vt_rep1, _yy_rep1, _xx_rep1);
223     _green_rep2 = _noise3_smp_16(_vt_rep2, _yy_rep2, _xx_rep2);
224     _green_rep3 = _noise3_smp_16(_vt_rep3, _yy_rep3, _xx_rep3);
225     _blue_rep0 = _noise3_smp_16(_yy_rep0, _xx_rep0, _vt_rep0);
226     _blue_rep1 = _noise3_smp_16(_yy_rep1, _xx_rep1, _vt_rep1);
227     _blue_rep2 = _noise3_smp_16(_yy_rep2, _xx_rep2, _vt_rep2);
228     _blue_rep3 = _noise3_smp_16(_yy_rep3, _xx_rep3, _vt_rep3);
229     _red_rep0 += ((const float __attribute__((vector_size(16)))){bias, bias, bias, bias});
230     _red_rep1 += ((const float __attribute__((vector_size(16)))){bias, bias, bias, bias});
231     _red_rep2 += ((const float __attribute__((vector_size(16)))){bias, bias, bias, bias});
232     _red_rep3 += ((const float __attribute__((vector_size(16)))){bias, bias, bias, bias});
233     _green_rep0 += ((const float __attribute__((vector_size(16)))){bias, bias, bias, bias});
234     _green_rep1 += ((const float __attribute__((vector_size(16)))){bias, bias, bias, bias});
235     _green_rep2 += ((const float __attribute__((vector_size(16)))){bias, bias, bias, bias});
236     _green_rep3 += ((const float __attribute__((vector_size(16)))){bias, bias, bias, bias});
237     _blue_rep0 += ((const float __attribute__((vector_size(16)))){bias, bias, bias, bias});
238     _blue_rep1 += ((const float __attribute__((vector_size(16)))){bias, bias, bias, bias});
239     _blue_rep2 += ((const float __attribute__((vector_size(16)))){bias, bias, bias, bias});
240     _blue_rep3 += ((const float __attribute__((vector_size(16)))){bias, bias, bias, bias});
241     _red_rep0 = __builtin_ia32_blendvps(_red_rep0,
242         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}),
243         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpgtps(_red_rep0,
244             ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}))))));
245     _red_rep1 = __builtin_ia32_blendvps(_red_rep1,
246         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}),
247         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpgtps(_red_rep1,
248             ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}))))));
249     _red_rep2 = __builtin_ia32_blendvps(_red_rep2,
250         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}),
251         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpgtps(_red_rep2,
252             ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}))))));
253     _red_rep3 = __builtin_ia32_blendvps(_red_rep3,
254         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}),
255         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpgtps(_red_rep3,
256             ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}))))));
257     _green_rep0 = __builtin_ia32_blendvps(_green_rep0,
258         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}),
259         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpgtps(_green_rep0,
260             ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}))))));
261     _green_rep1 = __builtin_ia32_blendvps(_green_rep1,
262         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}),
263         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpgtps(_green_rep1,
264             ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f}))))));
265     _green_rep2 = __builtin_ia32_blendvps(_green_rep2,
```

```

266         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f})),
267         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpgtps(_green_rep2,
268         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f})))));
269     _green_rep3 = __builtin_ia32_blendvps(_green_rep3,
270         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f})),
271         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpgtps(_green_rep3,
272         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f})))));
273     _blue_rep0 = __builtin_ia32_blendvps(_blue_rep0,
274         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f})),
275         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpgtps(_blue_rep0,
276         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f})))));
277     _blue_rep1 = __builtin_ia32_blendvps(_blue_rep1,
278         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f})),
279         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpgtps(_blue_rep1,
280         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f})))));
281     _blue_rep2 = __builtin_ia32_blendvps(_blue_rep2,
282         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f})),
283         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpgtps(_blue_rep2,
284         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f})))));
285     _blue_rep3 = __builtin_ia32_blendvps(_blue_rep3,
286         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f})),
287         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpgtps(_blue_rep3,
288         ((float __attribute__((vector_size(16)))){1.0f, 1.0f, 1.0f, 1.0f})))));
289     _red_rep0 = __builtin_ia32_blendvps(_red_rep0,
290         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})),
291         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpltgps(_red_rep0,
292         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})))));
293     _red_rep1 = __builtin_ia32_blendvps(_red_rep1,
294         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})),
295         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpltgps(_red_rep1,
296         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})))));
297     _red_rep2 = __builtin_ia32_blendvps(_red_rep2,
298         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})),
299         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpltgps(_red_rep2,
300         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})))));
301     _red_rep3 = __builtin_ia32_blendvps(_red_rep3,
302         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})),
303         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpltgps(_red_rep3,
304         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})))));
305     _green_rep0 = __builtin_ia32_blendvps(_green_rep0,
306         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})),
307         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpltgps(_green_rep0,
308         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})))));
309     _green_rep1 = __builtin_ia32_blendvps(_green_rep1,
310         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})),
311         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpltgps(_green_rep1,
312         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})))));
313     _green_rep2 = __builtin_ia32_blendvps(_green_rep2,
314         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})),
315         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpltgps(_green_rep2,
316         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})))));
317     _green_rep3 = __builtin_ia32_blendvps(_green_rep3,
318         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})),
319         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpltgps(_green_rep3,
320         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})))));
321     _blue_rep0 = __builtin_ia32_blendvps(_blue_rep0,
322         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})),
323         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpltgps(_blue_rep0,
324         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})))));
325     _blue_rep1 = __builtin_ia32_blendvps(_blue_rep1,
326         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})),
327         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpltgps(_blue_rep1,
328         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})))));
329     _blue_rep2 = __builtin_ia32_blendvps(_blue_rep2,
330         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})),
331         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpltgps(_blue_rep2,
332         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})))));
333     _blue_rep3 = __builtin_ia32_blendvps(_blue_rep3,
334         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})),
335         ((float __attribute__((vector_size(16)))) (__builtin_ia32_cmpltgps(_blue_rep3,
336         ((float __attribute__((vector_size(16)))){0.0f, 0.0f, 0.0f, 0.0f})))));
337     _red_rep0 *= ((float __attribute__((vector_size(16)))){255.0f, 255.0f, 255.0f, 255.0f});
338     _red_rep1 *= ((float __attribute__((vector_size(16)))){255.0f, 255.0f, 255.0f, 255.0f});
339     _red_rep2 *= ((float __attribute__((vector_size(16)))){255.0f, 255.0f, 255.0f, 255.0f});
340     _red_rep3 *= ((float __attribute__((vector_size(16)))){255.0f, 255.0f, 255.0f, 255.0f});
341     _green_rep0 *= ((float __attribute__((vector_size(16)))){255.0f, 255.0f, 255.0f, 255.0f});
342     _green_rep1 *= ((float __attribute__((vector_size(16)))){255.0f, 255.0f, 255.0f, 255.0f});
343     _green_rep2 *= ((float __attribute__((vector_size(16)))){255.0f, 255.0f, 255.0f, 255.0f});
344     _green_rep3 *= ((float __attribute__((vector_size(16)))){255.0f, 255.0f, 255.0f, 255.0f});

```

```

345     _blue_rep0 *= ((float __attribute__((vector_size(16)))){255.0f, 255.0f, 255.0f, 255.0f});
346     _blue_rep1 *= ((float __attribute__((vector_size(16)))){255.0f, 255.0f, 255.0f, 255.0f});
347     _blue_rep2 *= ((float __attribute__((vector_size(16)))){255.0f, 255.0f, 255.0f, 255.0f});
348     _blue_rep3 *= ((float __attribute__((vector_size(16)))){255.0f, 255.0f, 255.0f, 255.0f});
349     *((unsigned char __attribute__((vector_size(16)))) (*) &(output_red[(j * rowstride) + i])) =
350         __conv_float_to_uchar_smp16(_red_rep0, _red_rep1, _red_rep2, _red_rep3);
351     *((unsigned char __attribute__((vector_size(16)))) (*) &(output_green[(j*rowstride) + i])) =
352         __conv_float_to_uchar_smp16(_green_rep0, _green_rep1, _green_rep2, _green_rep3);
353     *((unsigned char __attribute__((vector_size(16)))) (*) &(output_blue[(j*rowstride) + i])) =
354         __conv_float_to_uchar_smp16(_blue_rep0, _blue_rep1, _blue_rep2, _blue_rep3);
355     *((unsigned char __attribute__((vector_size(16)))) (*) &(output_alpha[(j*rowstride) + i])) =
356         __conv_int_to_uchar_smp16(((int __attribute__((vector_size(16)))){255, 255, 255, 255}),
357         ((int __attribute__((vector_size(16)))){255, 255, 255, 255}),
358         ((int __attribute__((vector_size(16)))){255, 255, 255, 255}),
359         ((int __attribute__((vector_size(16)))){255, 255, 255, 255}));
360     }
361     for (; i < img_width; i++)
362     {
363         float vx, vy, vt;
364         float red, green, blue;
365         float xx, yy;
366         vx = i * vdx;
367         vy = j * vdy;
368         vt = time * vs;
369         xx = vx * vs;
370         yy = vy * vs;
371         red = noise3(xx, vt, yy);
372         green = noise3(vt, yy, xx);
373         blue = noise3(yy, xx, vt);
374         red += bias;
375         green += bias;
376         blue += bias;
377         red = (red > 1.0f) ? 1.0f : red;
378         green = (green > 1.0f) ? 1.0f : green;
379         blue = (blue > 1.0f) ? 1.0f : blue;
380         red = (red < 0.0f) ? 0.0f : red;
381         green = (green < 0.0f) ? 0.0f : green;
382         blue = (blue < 0.0f) ? 0.0f : blue;
383         red *= 255.0f;
384         green *= 255.0f;
385         blue *= 255.0f;
386         output_red[(j * rowstride) + i] = red;
387         output_green[(j * rowstride) + i] = green;
388         output_blue[(j * rowstride) + i] = blue;
389         output_alpha[(j * rowstride) + i] = 255;
390     }
391 }
392 }
393 }

```

Listing 21. Perlin Noise kernel: GCC vector extensions